AUTOMATION IN CS1 WITH THE
FACTORING PROBLEM GENERATOR

A Project
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Joshua B. Parker
December, 2009

© 2009
Joshua B. Parker
ALL RIGHTS RESERVED

Committee Membership

TITLE: Automation in CS1 with the Factoring Problem Generator

AUTHOR: Joshua B. Parker

DATE SUBMITTED: December, 2009

COMMITTEE CHAIR: Dr. Clinton Staley, Professor of Computer Science

COMMITTEE MEMBER: Dr. Aaron Keen, Professor of Computer Science

COMMITTEE MEMBER: Dr. Clark Turner, Professor of Software Engineering

Abstract

AUTOMATION IN CS1 WITH THE FACTORING PROBLEM GENERATOR

Joshua B. Parker

As the field of computer science continues to grow, the number of students enrolled in related programs will grow as well. Though one-on-one tutoring is one of the more effective means of teaching, computer science instructors will have less and less time to devote to individual students. To address this growing concern, many tools that automate parts of an instructor's job have been proposed. These tools can assist instructors in presenting concepts and grading student work, and they can help students learn to program more effectively. A growing group of intelligent tutoring systems attempts to tie all of this functionality into a single tool that is meant to be used throughout an entire CS course or series of courses.

To contribute to this emerging area, the Factoring Problem Generator (FPG) is presented in this work. The FPG creates and grades problems in C in which students search for and extract blocks of repeated code into individual functions, learning to utilize parameters and return values as they do so. The problems created by the FPG are highly configurable by instructors such that the difficulty can be finely tuned to suit students' individual needs. Instructors can choose whether or not to include arrays, pointers, certain elemental data types, certain operators, or certain kinds of statements, among other things. The FPG is additionally capable of generating a set of test cases for each generated problem. These test cases fully exercise students' solutions by covering all branches of execution, and they ensure that program functionality does not change as students factor code into functions.

Initial experimentation with the system has suggested that the FPG can be integrated into a beginning CS curriculum and with further refinement could become a standard tool in the CS classroom.

Keywords: function factoring, intelligent tutoring system, classroom automation, CS1, CS2, code generation, test case generation

Acknowledgements

I could not have completed this thesis without the help of my committee, so I owe many thanks to Drs. Aaron Keen and Clark Turner as well as to my adviser, Dr. Clint Staley. To Clint, I am deeply grateful for his willingness to work with me as a remote student and for his ideas, advice, and encouragement along the way.

I must also thank the staff and faculty of the Cal Poly Departments of Computer Science and Computer Engineering for their instruction and guidance in getting me to this point.

To my friends and family in the Cal Poly Bands and the Cal Poly Music Department, I offer my thanks for five years of memories and for five years of experiences that have made me more of a well-rounded individual.

Finally, my father, Jim; my mother, Charlotte; and my step-father, Dave, deserve my utmost gratitude and praise for the incredible amount of love, support, and guidance they have given me throughout my life. Everything that I am, I am because of them.

– J.B.P.

Table of Contents

List of Tables	vii
List of Figures	viii
1. Introduction.....	1
2. Related Works.....	3
2.1. Visualization Tools	4
2.2. Assessment Tools.....	10
2.3. Programming Environments	15
2.4. Intelligent Tutoring Systems	19
3. The Intelligent Homework System and its Motivation.....	23
3.1. The Stair-Step Model for Teaching STEM Subjects	23
3.2. The Intelligent Homework System	27
4. The Factoring Problem Generator from the Users' Perspectives	34
4.1. Motivation Behind the FPG	34
4.2. FPG Problems from the Student's Perspective	35
4.3. Adjusting FPG Problems as an Instructor.....	40
5. Implementation of the Factoring Problem Generator	47
5.1. Creating Function Headers and the Function Call Graph.....	48
5.2. Filling in Functions with Meaningful Code.....	52
5.3. Inlining the Solution into the Problem.....	62
5.4. Creating Test Cases.....	68
6. Experiments and Experimental Data	77
6.1. Manual Examination of FPG Problems	77
6.2. Student Use of the FPG Module	81
7. Summary and Conclusions	84
7.1. Contributions.....	85
7.2. Future Directions	86
Bibliography	89
Appendix – FPG Feedback Survey.....	92

List of Tables

Table 3-1: Summary of Major IHS Classes and Interfaces	29
Table 3-2: Methods in the Grader Interface.....	31
Table 4-1: Summary of Configurable Values in an FPG Parameter	41
Table 4-2: Types of Statements Used in Programs Generated by the FPG	44
Table 5-1: Classes of Variables Supported by the FPG.....	51

List of Figures

Figure 3-1: Understanding and Morale versus Effort in a Non-STEM Subject	24
Figure 3-2: Understanding and Morale versus Effort in a STEM Subject	26
Figure 3-3: Diagram of an IHS Attempt	30
Figure 4-1: FPG Problem as Presented to a Student.....	37
Figure 4-2: Unfactored Program (left) Compared to its Factored Solution (right).....	38
Figure 4-3: Effect of the maxOpsInParamExpr Parameter Value on Difficulty	45
Figure 5-1: Process of Creating the Function Call Graph.....	49
Figure 5-2: Example of Meaningless Code	52
Figure 5-3: Process of Creating Non-Branching Code	55
Figure 5-4: Process of Creating Branching Code	58
Figure 5-5: Loop Containing Meaningless Code.....	61
Figure 5-6: The Simple Inlining Process	62
Figure 5-7: The FPG Inlining Process	64
Figure 5-8: The Actual Parameter Selection Algorithm.....	65
Figure 5-9: Finding Starting Conditions through Backward Code Computation.....	69
Figure 5-10: Maintaining States as the FPG Creates Statements	73
Figure 6-1: Function Containing an Extra Variable	78
Figure 6-2: Example of Easily Simplified Code.....	79
Figure 6-3: Example of Combinable Loops.....	80

1. Introduction

For many years, computer science instructors have been attempting to put the tools of their trade to good use in the classroom. Indeed, it was found that more than one-fifth of papers presented at conferences of the ACM Special Interest Group for Computer Science Education over the last few decades have presented automated tools serving both students and instructors in the field of computer science (Valentine 2004). With large numbers of students enrolling in CS programs, tools of this kind are in high demand. Unfortunately, few of them have come to see anything other than entirely local use (Pears 2007).

It has also been observed that one of the more difficult concepts for beginning CS students to learn is separating the functionality of their programs into individual procedures (Lahtinen 2005). The procedure, or *function* as it would be called in the C programming language, is one of the most basic but useful tools available to programmers and is one that is essential for beginning CS students to master.

It was for these reasons that the subject of this report, the Factoring Problem Generator (FPG), was created. The FPG is a tool that instructors can use to automatically create and grade programming problems in which students extract blocks of common code into individual functions, a process called *factoring*. Through this process, students can gain experience in dividing functionality into compartmentalized subroutines as well as in utilizing parameters and return values.

The FPG additionally provides a high degree of customization such that instructors can tailor FPG problems to suit the needs of students. The system is thus adaptable to any computer science classroom. Its automation allows instructors to assign unique problem sets to individual students without having to create and grade each problem, and it contributes to the growing field of automation in computer science education.

The rest of this report is organized as follows. A survey of related works in CS education is presented in Chapter 2, concluding with an exploration of the direction the field has taken over the past few years. Chapter 3 provides an overview of the automated tutoring system into which the FPG fits, the Intelligent Homework System. The users' views of the FPG system are presented in Chapter 4, including both a walkthrough of how a student might solve an FPG problem and an overview of the problem customization options available to instructors. Chapter 5 presents the detailed implementation of the FPG system, covering both the process through which problems are created and the process through which test cases to verify student solutions are generated. A summary of findings from experiments evaluating the FPG is given in Chapter 6. Finally, the report concludes in Chapter 7 and outlines possible future directions of the FPG system.

2. Related Works

To address the growing concern of the burden on computer science instructors, many tools that automate parts of an instructor's job have been proposed. These tools attempt to assist educators in communicating technical concepts to students, and they strive to simplify and reduce an instructor's workload.

A survey conducted of papers presented at SIGCSE Technical Symposium conferences between 1984 and 2003 found that 22% of the papers presented software tools aimed at helping instructors and/or students (Valentine 2004). Though there has been a great deal of research performed in this area, a very small percentage of these tools have been used outside of the institution where they were developed (Pears 2007).

There are three principal factors that have contributed to this trend. First, most educational tools are developed to solve a local problem, and so multiple tools have been developed that attempt to solve similar problems. Second, these tools are often developed as part of doctoral research and are completely unsupported when the developers graduate. Finally, there is little funding available to implement features of tools that would meet the needs of the larger set of instructors and students. Because of this, tools are commonly inflexible and do not support modification (Pears 2007).

In spite of this, a small number of automated tools have been fairly widely adopted and have met with noteworthy success. They can be loosely categorized into visualization tools, assessment tools, and programming environments. In this chapter, each of these

areas and the major successful works that are a part of them are examined. Additionally, the emerging concept of an intelligent tutoring system is presented along with some examples of such a system.

2.1. Visualization Tools

Human beings are good at processing visual information. However, most programming features, algorithms, and data structures are abstract ideas that are not easily translated into a graphical form. Additionally, these ideas are often dynamic in nature and are thus not easily understood by the novice programmer (Pears 2007). Because of this, a great deal of research has been applied to the visual presentation of programming structures, code execution, and algorithms.

Visualization tools are those that assist computer science educators in presenting programming material in graphical forms. These tools can be generally classified in one of two areas. Code visualization tools, such as Jeliot (Moreno 2004) or jGRASP (Jain 2006), visualize the execution of actual blocks of code and are useful for helping a student to understand how a program he has coded functions line by line. These are often associated with certain programming environments, discussed later in this chapter. On the other hand, algorithm visualization tools like JHAVE (Naps 2005) and MatrixPro (Karavirta 2004) focus on presenting algorithms or more abstract concepts, and they may omit code altogether. This section first examines what it takes for the general visualization tool to be effective. It then presents a summary of research currently being conducted in both of the subcategories of visualization tools.

2.1.1. Characteristics of Effectiveness of Visualization Tools

In any visualization tool, it is important to craft the tool in such a way that it is engaging to the student. Naps states in (Naps 2002) that any tool, “no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity.” Visualization tools cannot simply present animations of algorithms and expect to keep a student’s attention. According to Naps’s work, there are four active areas of engagement:

- Responding
- Changing
- Constructing
- Presenting

Any visualization tool that is to be effective must engage the student in one of these four activities. This is verified in (Hundhausen 2002), where 83% of visualization tools that “actively engage the student” – as compared to 33% of tools that only display “visualizations watched by the student” – were declared effective.

2.1.2. Code Visualization Tools

Code visualization tools can illustrate the static data structures associated with a particular program or the dynamic aspects of executing actual program code. This subsection examines three such tools.

Jeliot 3 was released in 2004 by Moreno et al (Moreno 2004). Jeliot is a programming environment used specifically to illustrate object-oriented programming concepts in Java,

and it is the third generation of the tool. In Jeliot, students can write code in Java and watch the visualizations of each of their objects – complete with member data – evolve as their program executes. The user interface allows the student to fully control the program execution step by step; it allows play, pause, and even rewind actions while viewing code’s effects on objects. Moreno et al are careful to keep all animations consistent, as they note that novices do not have the skills to understand the “secondary notation” of the visualization (Moreno 2004). The developers are extending Jeliot to be able to be used past the first few weeks of programming instruction, but their presentation of Jeliot does not present any experiment to speak to the effectiveness of the tool.

Another such code visualization tool is jGRASP, released in 2006 by Jain et al at Auburn University (Jain 2006). The primary feature of jGRASP is the “object viewer” that generates synchronized, dynamic, state-based visualizations of both student-programmed objects and primitive variables in Java. jGRASP is substantial enough as an IDE to be used throughout an entire introductory programming course series. The developers report “positive feedback” from instructors using their tool, and they show that the use of jGRASP decreased student time to program an acceptable solution to a CS2 lab problem and increased the average score received on the assignment.

JHAVE and jGRASP were designed specifically for the beginning programmer, but visualization tools can be applied in more than just the introductory programming courses. Visual debuggers are tools that assist programmers in removing programming bugs from their code using a graphical interface. The Data Display Debugger (DDD), presented by Zeller in 1996, was designed to compete with commercial debuggers (Zeller

1996). DDD presents a “graphical data display” in which the user may explore complex data structures at a breakpoint in the code through simple mouse clicks. These features, however, are likely to be too complex to be understood by a novice programmer and thus are of limited use in CS1 and CS2.

It is seen that object-oriented languages like Java are more easily transferred into a graphical context than are procedural languages like C. However, object-oriented languages are not necessarily the languages of choice for introductory programming courses; in fact, the language debate for CS1 and CS2 has been ongoing for four decades (Pears 2007). Useful and beginner-friendly code visualization tools that are applicable to languages other than Java (and other object-oriented languages) will see use by the sect of instructors that believe in a “procedural language first” curriculum.

2.1.3. Algorithm Visualization Tools

Whereas code visualization tools are more useful to the student in writing his own programs, algorithm visualization (AV) tools are more useful to the student in learning a new concept for the first time (or to the instructor teaching that concept). These tools can be used by the instructor lecturing in the classroom, or as secondary instruction by students studying on their own outside of class.

Systems have been proposed that allow instructors to create their own algorithm visualizations. The Java and Web-based Algorithm Animation (JAWAA) tool, proposed by Akingbade et al at Duke University in 2003, is one such system (Akingbade 2003). JAWAA is a scripting language that allows instructors to code the visualization of an

algorithm on a number of predefined data structures (like arrays, stacks, trees, and lists) with certain predefined operations (like change value, change color, and swap). Instructors must program their own visualizations in JAWAA, but they can be saved and reused, and shared with other instructors as well.

The Java-Hosted Algorithm Visualization Environment (JHAVE), proposed by Naps in 2005, is another such system (Naps 2005). JHAVE is a support environment coded in Java that anyone can use to create their own AV tools. It provides a number of different features, including a standard set of “VCR” controls for watching the visualizations, input generators, information and pseudocode windows that give additional information on the algorithm, and an interface for programming “stop-and-think” questions that actively engage the student. JHAVE is not an AV system in itself, but rather it provides the common functionality that instructors can use to create their own tailored AV applications.

The flexibility that JAWAA and JHAVE provide comes at a cost: that instructors must use the tools supplied to create their own systems. The MatrixPro system, proposed by Karavirta et al in 2004, takes a different approach to providing flexibility (Karavirta 2004). The user of MatrixPro does not need to write any code to produce a visualization. Instead, the user can invoke a number of provided operations on the included data structures through the graphical interface. This allows instructors to easily create on-the-fly simulations of an algorithm, and they can even modify them in class to deal with the “what if” questions that may arise during a lecture. Because the system requires no coding, even beginning programmers can use it to present concepts they have learned.

Allowing students to create their own animations is another avenue of research in this field. The Algorithm Visualization Storyboarder (ALVIS) was developed by Hundhausen et al in 2000 to enable students to easily create “low fidelity” animations (Hundhausen 2000). ALVIS allows students to draw their own figures and create primitive animations with them in order to present their own knowledge of algorithms to their peers and their instructors.

Krebs et al took this approach one step further in 2005 with their Multimedia Algorithms and Data Structures Assessment (MA&DA) system (Krebs 2005). As with ALVIS, MA&DA allows students to construct their own visualizations from scratch, but these visualizations must be of a predefined set of algorithms or data structures. MA&DA is then able to provide immediate assessment and feedback to the student as the animation is created. The degree of this feedback, however, is variable and can be turned off completely to meet the student’s needs. Krebs et al propose to extend this automatic assessment to a more general animation tool, similar to the one ALVIS provides, that allows students to create their own images from a digital “white board.”

The tools presented in this section can be grouped into two sets. One set provides tools for instructors to more easily code their own visualizations, but this often requires a great deal of effort on the part of the instructor. The other set provides ready-made functionality, but only a select number of operations are available for a select number of data structures. The happy medium lies in the middle of these two, so researchers may look to lessen the work required for instructors to code their own simulations, or they may look to expand and generalize the capabilities of the ready-made solutions.

2.2. Assessment Tools

Automated assessment tools are useful for a number of reasons. They allow instructors to streamline the grading process so more of their time can be spent helping students with more substantial issues than the administrative overhead of assigning grades. Assessment tools also allow students to submit their work at any time and receive immediate feedback, the majority of which can be tailored to explain the most common problems encountered by students in any particular program. As the number of students enrolling in programming courses and requiring one-on-one instructional time with computer science educators increases, the automation of assessment tools will become more and more important.

A number of successful assessment tools and the approaches they take to evaluating students' performances are presented in this section. Additionally, this section explores two particular problems in the area: automatic generation of test cases and plagiarism detection.

2.2.1. General Assessment Tools

The BOSS assessment tool was initially developed at the University of Warwick in 1998, and since then a number of different versions of the system have been produced (Joy 2005). Originally, BOSS presented a set of command-line tools, such as a tool for checking a programmed solution against a set of hidden test cases and a tool for submitting code to a secure server for grading. The later version of BOSS transferred the command-line functionality to a graphical interface and added features such as plagiarism

detection and a web server component that allows multiple tutors to review submissions using a web browser (Douce 2005). BOSS is representative of many of the assessment tools that have been developed.

The CourseMaker system provides more functionality than does BOSS. CourseMaker (Higgins 2005) evolved from the Celedith system, developed at Nottingham University in the mid-eighties. CourseMaker allows the same assignment-specific evaluation that BOSS can perform, but the feedback is more specific in that it allows students to see the specific areas where points were lost. The system also provides statistical data that can give students a clearer picture of their performance against the rest of the class. The most noteworthy feature of CourseMaker, however, is that it allows the organization of multiple assignments and other educational documents into courses. Students, instructors, and tutors each have a login to the system and can be assigned differing permissions. Thus, the system is intended to be used at a university-wide scale (Douce 2005).

As more universities begin to teach in languages like Java, it will become more common for beginning CS students to write GUI-based programs rather than command-line-based programs. The JEWL system was developed at the University of Brighton to address this concern (English 2004). Through extensions to the commonly-used graphical interface elements provided in the Java libraries (like text boxes, buttons, and text boxes), JEWL can automatically run a GUI-based program with a given set of inputs to determine the program's correctness. The set of programs that can be tested using JEWL is limited by the range of GUI elements extended by the system. Additionally, JEWL only tests for

input/output behavior and does not assess the quality of the user interface. Evaluating the interface of GUI-based programs remains an unsolved problem.

Though these assessment tools are useful, the immediate feedback they can provide may encourage some students to take a “trial-and-error” approach to programming, making blind tweaks to programs until the program passes all tests. The Web-based Center for Automated Grading (WebCAT) system, created at Virginia Tech in 2003, addresses this concern in a unique way (Edwards 2003). WebCAT requires students to submit test data along with their code. Students then not only receive a grade for their program, but also a grade for the correctness of their test cases and the extent to which the test cases cover the stated problem. As test-driven development becomes more commonplace in the classroom, systems like WebCAT need to be researched more thoroughly.

Finally, assessment tools may be extended to cover more abstract assignments. The aforementioned CourseMaker tool supports the submission of flowcharts. These flowcharts are translated into actual code and assessed by running the standard breed of input/output comparison tests on them (Higgins 2005). Another tool for evaluating abstract assignments is the TRAKLA2 system, which was designed specifically for assessing students’ understanding of algorithms by allowing them to simulate the algorithm using direct manipulation (Malmi 2004). TRAKLA2 presents students with randomly generated inputs and asks students to perform different steps of an algorithm with them. This breed of assessment system is less common but not necessarily less valuable.

Most of these tools compare a program's output to the instructor-specified "correct" output. A portion of them have implemented more advanced metrics to assess programs, like program efficiency or complexity (Pears 2007). These metrics, however, are not as refined and remain an open area of research. Additionally, (Douce 2005) states that it may be valuable to evaluate code design, which may involve taking names of functions and variables into account as well as examining comments within submitted programs.

2.2.2. Automated Generation of Test Cases

The assessment systems that are currently available today all revolve around the same idea: instructors first specify a problem and the tests that should be used to measure a student's solution to that problem, and then students can submit their programs for automatic grading. These tools save instructors from having to grade each student's solution by hand. However, the specification of a complete and correct set of test cases is a nontrivial issue and is currently one left up to each user of any of the assessment systems.

Research has been done investigating the automated generation of test cases, although none with respect to CS1-level programs. Cohen et al has proposed the AETG system for using combinatorial logic to cover all test cases (Cohen 1997). His system, however, still requires some degree of input from the user. Colin et al has presented two algorithms for generating test cases based on boundary conditions (Colin 2004), but this requires specifying the problem in a complex set of computational models. Neither of these groups applies their solution specifically to automated testing in the classroom.

The problem of instructors having to specify their own test cases rather than, for example, having them generated automatically from the instructor's solution program, is an unsolved problem. The automated generation of test cases for use with assessment tools, then, remains an open area of research.

2.2.3. Plagiarism Detection

Plagiarism is a frequent concern, especially among instructors of introductory programming courses. Some of the aforementioned assessment systems include plagiarism detection tools, such as the BOSS system (Joy 2005). Other tools have been developed for use specifically to detect plagiarism.

The MOSS (Aiken n.d.), YAP (Wise 1996), and JPlag (Malpohl n.d.) tools were all developed specifically to provide an easy way to measure the similarity between submitted programs. In general, these programs work by dividing code into sequences of tokens and then comparing those sequences against other programs' using a variety of metrics to pick out the cases that were most likely plagiarized. These programs have all seen fairly widespread use at a number of universities, and case studies have been presented that attest to their effectiveness (Chen 2004).

The Software Integrity Diagnosis (SID) tool, developed at the University of California, Santa Barbara in 2004, takes a slightly different approach to detecting plagiarism (Chen 2004). SID applies an information-based metric that can be applied to any sequence (e.g. DNA or English sentences) to programs to determine the degree of similarity between them. The authors of SID present a number of case studies from using their tool in their

own institution, and they find that SID performs as well as MOSS, YAP, and JPlag, but it does not necessarily outperform them.

2.3. Programming Environments

All programmers – not just CS1 students – tend to work within development environments that provide the tools necessary to accomplish programming tasks. The simplest of these environments would be a simple text editor (for writing source files) and a compiler. Most programmers, however, prefer to work in more advanced integrated development environments (IDEs) that provide a wealth of features in addition to the basics. According to (Pears 2007), the most significant of these features are:

- Organization of program components into projects
- Language specific editing features (like syntax highlighting and code completion)
- Support tools (like visual debuggers, testing tools, and documentation generators)
- Support for almost any widely-used programming language.

The complex set of features provided in a professional IDE, however, is often too much for a novice to learn. Because of this, a number of programming environments have been developed specifically for the beginning programmer. These can be generally categorized as either programming support tools or microworlds. Programming support tools present standard execution environments in which students can create programs, and they often have a simpler toolset than their professional counterparts. Microworlds provide environments based on physical entities to lessen the gap between students' mental models and the programming language (Pears 2007). This section examines the major programming environments in each of these categories.

2.3.1. Programming Support Tools

In this context, a programming support tool is an environment designed specifically for the beginning programmer. Some of these tools, like GILD (Storey 2003), are simply modified versions of professional IDEs that hide some of the more advanced features, lessening the learning curve of the environment. Others use specially designed features to promote the learning of particular paradigms of writing programs.

The first of these are BlueJ, a project proposed in 2003 (Kölling 2003), and DrJava, developed at Rice University in 2002 (Allen 2002). Both of these Java development tools allow students to instantiate objects, call methods on them, and evaluate expressions with them. (Pears 2007) terms this “interactive incremental code execution”. Tools like these allow students to interact with and explore the objects they create without having to write all of the code necessary to do so.

Additionally, BlueJ automatically generates a static visualization of the interactions between the user-defined objects, showing inheritance and ownership relationships. This kind of visualization can be useful to the student, but it has been suggested that static visualizations need to be augmented with dynamic visualizations if they are to be useful (Ragonis 2005). The jGRASP tool (Jain 2006) is perhaps the best example of such dynamic visualization. It provides students with the “object viewer” that can be used to view the states of objects as the code is incrementally executed. Other code visualization tools have been discussed earlier in this chapter.

Finally, programming support tools may provide features for automatically editing or even generating program code. Such a tool is JPie, created at Washington University in St. Louis (Goldman 2004). JPie abstracts away the actual code by representing class definitions graphically and allowing them to be created and modified through the graphical user interface. In using JPie in Washington University's introductory programming courses and recording end-of-term student evaluations, the developers were able to show that the practice of defining objects to solve assigned problems without having to worry about code was an enjoyable experience for students.

All of these programming support tools are targeted at Java. This is perhaps because object-oriented languages lend themselves to more compartmentalized representations that can be easily manipulated through a user interface. Many computer science educators, however, question whether the move to the "objects-first" curriculum is justified and continue to teach CS1 using a procedural language (Pears 2007). The direction of research in programming support tools suggests that these instructors will not likely be supported with new and evolving tools that they may use in their classrooms.

2.3.2. Microworlds

A microworld presents a physical metaphor (often in the form of a visualization on a computer screen) that intends to lessen the distance between the programming language and the mental models held by the students (Pears 2007). Two microworlds are examined in this subsection.

The Karel system was originally proposed in 1981 to be used as the first part of a CS1 course (Pattis 1981). The system used its own language to control robots living in a world of streets and intersections. Karel allowed instructors to teach the fundamentals of translating a problem's solution into code while using a language with simplified syntax for beginners. In 2001, Karel was redesigned to use an object-oriented paradigm, and the new system was renamed Karel++ (Becker 2001). Again, the system encouraged the translation of a solution into code, and it made the transition to actual object-oriented programming more seamless. Karel++ has been used in classes at the University of Waterloo, and the instructors report positive experiences with the tool.

Expanding on this concept is Alice, a tool developed at Carnegie Mellon University as part of a pre-CS1 course (Cooper 2003). The graphical world of Alice is populated with various 3D objects such as people, animals, and vehicles. As in Karel, these objects are defined and controlled with methods similar to the ones that would be found in an object-oriented language. Unlike Karel, however, Alice allows the control scripts to be created through the user interface by dragging and dropping the instructions that manage the world. The visual feedback presented by Alice allows programmers to see how programming statements execute, preparing them to write their own programs in CS1.

The relative success of Alice suggests a new avenue of research: developing systems to teach programming concepts to students before they take their first CS course in college or even in high school. Preparing children early for advanced study of computers may become a mainstay in the primary education curriculum.

2.4. *Intelligent Tutoring Systems*

The areas of the aforementioned automated tools have been relatively well explored. The future of automation in CS1 instead lies in the area of intelligent tutoring systems. An *intelligent tutoring system* combines elements from all of the aforementioned tool classifications – visualization tools, assessment tools, and programming environments – into a larger suite of software intended for use throughout an entire course or even series of courses. This section first presents the general characteristics of an intelligent tutoring system. It then examines a few case studies where tutoring systems have been successful. Finally, it presents the research being done on the frontier of this field and suggests the future directions the field may take.

2.4.1. *General Characteristics of Tutoring Systems*

According to (Pillay 2003), the integral features of an intelligent tutoring system (ITS) are:

- Presenting and explaining programming concepts
- Creating and assigning problems for students to work through
- Assisting students to develop solutions to assigned problems
- Evaluating student solutions with respect to correctness and efficiency
- Assisting students to debug semantic errors in their programs

A number of systems have been presented, such as (Anderson 1986) and (Soh 2006), that implement these features and that have met with positive reviews and increased student performance. However, no ITS has seen anything other than entirely local use. (Pillay 2003) suggests a number of reasons for this, among them that tutoring systems are

expensive to develop, that the systems that have been developed lack sharable components, and that tutoring systems have been both language- and platform-dependent and do not facilitate reusability.

2.4.2. Tutoring System Case Studies

One of the earliest successful tutoring systems was the Lisp Tutor, presented at Carnegie Mellon University (Anderson 1986). Through a textual user interface, the Lisp Tutor presents predefined problems to students and walks them through developing solutions to them by asking guiding questions and correcting students where they go wrong. The system is limited by the number of predefined problems programmed for it, and it is also one of the earliest cases to expose the problem of encouraging the “trial-and-error” approach to programming when giving immediate automated feedback. However, the instructors at Carnegie Mellon report increased student performance after using the tutor.

More recently, the ELM-ART tutor was developed to improve on the Lisp Tutor’s success (Weber 2001). ELM-ART provides a web-based graphic interface that the authors compare to a “digital textbook” that presents lectures, animations, problems, and assessments of student solutions. The major contribution of this work to the field was the NetCoach tool that dramatically increases the system’s flexibility by allowing instructors to construct course material for ELM-ART without having to program anything. The authors present a summary of student surveys that indicate the success of the system.

In 2005, Kumar presented another tutor used at Ramapo College of New Jersey (Kumar 2005). This tutor provides a lot of the same functionality of ELM-ART, but it also

provides a significant additional feature: it is capable of generating problems. Working from instructor-provided templates, Kumar's tutor generates random, unique blocks of code and asks students to evaluate how they will operate. Though the tutor does not provide a means for students to write their own programs, Kumar found through a series of quizzes administered before and after the use of his tutor that students who had worked with the tutor were better able to answer programming questions.

In 2006, Soh proposed the ILMDA system implemented at the University of Nebraska (Soh 2006). The ILMDA system added a new feature to the mix: the ability to "learn" from students. Using the statistical data collected through evaluation of the answers students give to the tutor's questions as well as through exam data supplied by the instructor, the ILMDA system selects questions for a student from areas in which that student needs more practice. Unlike Kumar's system, however, ILMDA draws from a pool of predefined questions. In spite of this, a study of the performances of two lab sections using the tutor and one section as the control group showed that ILMDA increased students' scores in the class.

2.4.3. Future Research in Tutoring Systems

A few significant studies have been performed that quantify certain characteristics of how students learn and that apply those findings to tutoring systems. From these, the direction that future tutoring systems may take can be inferred.

A considerable percentage of human communication is nonverbal. In 2005, Dadgostar et al performed a study on a group of students in primary school (Dadgostar 2005). By

recording students on video as they provided verbal responses to technical questions, the authors determined the degree to which students use hand gestures when speaking. They placed their findings in the context of tutoring systems by stating two design implications that should be considered: first, that the use of hand gestures must be considered in conjunction with the student's skill level; and second, that hand gestures must be interpreted as motion rather than as still images, so processing of them must be done in real time. The future of intelligent tutoring systems, as suggested by this study, involves reading the nonverbal cues from students.

Baker performed another study of students using a tutoring system at the University of Nottingham (Baker 2007). He collected a pool of data including the actions students took in the tutor, the time it took them to take those actions, their responses to the questions posed by the tutor, and other data. From this, he has proposed a model for determining when students are "off task" that can be used in the development of future intelligent tutoring systems. Baker also proposed ways in which future systems can respond to students that are determined to be off task.

Finally, the automatic generation of problems and the ability to assess them is an outstanding feature of the tutoring system presented by Kumar in (Kumar 2005). However, this is limited to asking students to trace blocks of code. It may be very useful to develop a system in which problems that students must produce code to solve are automatically generated and assigned.

3. The Intelligent Homework System and its Motivation

The previous chapter examined the field of automation in computer science education. It was seen that the proposed automated tools can be loosely classified into one of three categories: visualization tools, assessment tools, and programming environments. To answer the increasing demand for automation in the classroom, a more recent trend of developing what have been termed *intelligent tutoring systems*, or systems that combine elements from each of these classifications into a larger suite of software, has arisen. Though some prototype systems have been introduced, none have seen much more than entirely local use.

Dr. Clinton Staley, Professor of Computer Science at Cal Poly as well as the adviser of this work, has created a new intelligent tutoring system: the Intelligent Homework System (IHS). The Factoring Problem Generator (FPG) that is the subject of this work fits into the IHS framework, and so it is necessary to briefly describe the system before delving into the FPG. This chapter examines the major components of the IHS framework as well as one of Dr. Staley's teaching philosophies that has influenced the design of this new system.

3.1. The Stair-Step Model for Teaching STEM Subjects

Dr. Staley has observed that teaching a STEM (science, technology, engineering, mathematics) subject is different than teaching any other. He has quantified this difference by relating a student's effort both to the amount of understanding he gains and

to his morale during the process (Staley 2009). This section examines that relation in detail.

Consider a student of a non-STEM subject such as history as he studies for an upcoming exam. As the student studies his textbook and his notes from lecture, he memorizes the facts he must be able to recall for the exam, things like “The Declaration of Independence was signed on July 4, 1776.” If he studies for twice as long, he memorizes twice as many facts. As he learns more and more of the information he needs to pass the exam, his morale improves because he can see his own progress. A chart of his understanding and morale versus his effort appears in Figure 3-1.

Figure 3-1: Understanding and Morale versus Effort in a Non-STEM Subject

Compare this to a student studying a STEM subject. The things he must learn are not compartmentalized into simple facts (like the date of the signing of the Declaration of Independence). Instead, he must learn processes and how he might apply them in different situations. Take, for example, a calculus student studying differentiation. The simple fact that the derivative of x^2 is $2x$ will not help the student on an exam when he is asked to differentiate $4x^3$. The student must instead have learned *how* the process of differentiation is applied to an expression so that he can apply it himself to other expressions. While a student in a non-STEM subject may learn hundreds of facts in a course, a student may learn only a handful of processes in a STEM subject in the same time.

Learning these kinds of processes is not the same as memorizing facts. If it takes a student ten hours of studying to be able to differentiate any expression, he cannot study for only five hours and be able to differentiate half of all expressions. The student must put in all ten hours of studying, and then the rush of understanding of differentiation will tend to hit him in an “ah ha” moment. For this reason, the relationships among the student’s effort, understanding, and morale are very different for a STEM subject. See Figure 3-2.

Figure 3-2: Understanding and Morale versus Effort in a STEM Subject

Consider again the student trying to learn differentiation. As he attends lectures and reads his text book, he initially receives no new understanding of the process. He has put in effort but received no payoff, so his morale declines. As he continues to pour effort in, his morale sinks lower and lower. When the student finally achieves the “ah ha” moment and his understanding increases, his morale peaks and he moves on to learning the next process.

Instructors of STEM subjects can ease the students through this process by breaking up the “step” between being introduced to a process and being able to apply that process universally into several smaller steps. If, for example, a calculus instructor assigns problems in such a way that students achieve several small milestones along the way –

say, understanding the power rule and then understanding trigonometric differentiation – the students will see *some* payoff for their effort and morale will not decrease as drastically. If the large step is broken into several smaller steps, the relationships among effort, understanding, and morale for a STEM subject begin to look more like those for a non-STEM subject.

It should be noted that Dr. Staley’s model is based on his own learning and teaching experience and not on the collection and analysis of empirical data. Indeed, many non-STEM subjects include elements that are much more complex than the straight memorization of facts, and these are likely to exhibit the properties of a STEM subject as described above. This distinction, however, is not the point of this discussion. Rather, the point is that an instructor of *any* subject in which students must learn complex processes should strive to break up large steps into smaller ones such that students have smaller obstacles to overcome.

3.2. The Intelligent Homework System

The stair-step model holds true in teaching computer science. Consider a CS1 course where the curriculum consists of basic input and output, if/else statements, loops, functions, strings, and pointers. An instructor could give students one programming assignment for each of these topics, but the gap between, say, seeing an instructor write a while loop on the board and actually using one in a large program could be a difficult one for beginning CS students to cross. Instead, these tasks should be broken up into smaller milestones. Instructors, however, may find it difficult to create many new programming

problems, help students find solutions to them, and grade their responses, particularly when class sizes are large.

This is the major problem the Intelligent Homework System seeks to solve. Through automation, instructors need not be burdened by the creation and grading of programming assignments. The IHS even seeks to provide much more intelligent feedback than what platforms meant for seasoned developers provide so that students can receive assistance without having to wait in line for an instructor's office hour. In this section, the IHS framework is briefly described so that the reader may better understand how the FPG fits into it.

3.2.1. Overview of the IHS

The Intelligent Homework System is a framework into which fit individual modules for creating specific kinds of problems. The modules are responsible for the details of generating and grading their problems while the IHS takes care of general tasks such as communicating with students and keeping track of student progress. The IHS is implemented in Java, and its major components are summarized in Table 3-1.

Table 3-1: Summary of Major IHS Classes and Interfaces

Class/Interface	Description
Grader	The principle interface an IHS module must implement. Provides methods to create new problems and grade assigned problems.
GraderInfo	Provides general parameters from the IHS to the module, such as locale, problem seed, and state information.
Parameter	Provides module-specific parameters for constructing and grading problems.
GraderResult	Result of a Grader creating or grading a problem. Contains the created problem, scoring information, and messages for the student.
Responder	Interface through which an IHS module responds to its own problems, primarily to give correct solutions.
UIAtom	Abstraction of communication between an IHS module and a student. The IHS chooses how to display these, e.g. through a webpage or in a text-only format.

In general, a problem's life cycle begins by the IHS requesting a Grader to create a new problem. The returned problem is then presented to the student, and Grader-supplied state information is saved by the IHS. Because a student may be allowed any amount of time in which to solve a problem, a Grader implementation cannot assume that the object that created the problem will be the one to grade it. Graders, then, must assume that they cannot save their own states between method calls and instead pass all state information to the IHS for safekeeping.

When the student responds to the problem, the IHS passes the response and the state information back to the Grader. This may conclude the problem's life cycle, in which case the Grader returns a final score to the IHS. An IHS problem, however, may also consist of several smaller problems each requiring a student response, making up a longer conversation between the Grader and the student. If this is the case, the Grader may return a new problem to the IHS rather than a score (or, indeed, it may return the same

problem if the student's response was incorrect). One of these smaller problems, or a prompt requiring a single student response, is termed a *query* by the IHS. The longer conversation is termed an *attempt* by the student to solve the larger problem, and one attempt may consist of one or more queries. A diagram of one attempt is shown in Figure 3-3.

Figure 3-3: Diagram of an IHS Attempt

As an example, consider a Grader implementation that creates and grades addition problems. On initialization, the Grader creates a query that asks “What is $2 + 2$?” The student responds “5.” The Grader, seeing an incorrect response, returns a new query: “Sorry, that answer is not correct. What is $2 + 2$?” This time, the student answers “4.” The Grader sees the correct response and decides to push the student further by returning the new query “Correct! What is $14 + 23$?” The student answers “37.” The grader sees another correct response and ends the attempt by providing a final score to the IHS.

3.2.2. Detailed Design of the IHS

The Grader interface specifies the basic functionality that an IHS module must provide.

Its methods are listed in Table 3-2.

Table 3-2: Methods in the Grader Interface

Method	Description
initQuery	Creates the first query of an attempt using a GraderInfo object, which contains a Parameter and a random seed, among other things.
grade	Returns either a score for the attempt or a new query based on the student's response to the last query.
getResponder	Creates a Responder that can correctly or incorrectly respond to the current query.
getSampleParameter	Creates a "cookie cutter" parameter at one of ten difficulty levels that can be used to create a new attempt with initQuery.
getSupportedLocales	Returns the list of locales supported by the Grader.

The initQuery method is called to start a new attempt. The GraderInfo object passed to it contains a Parameter and a random seed, among other things. The Parameter class is defined by the Grader implementation, and it contains any parameters needed to configure the Grader to create and grade a problem. A specific instance of a Parameter coupled with a specific seed always results in the same problem produced by the Grader. The initQuery method returns a new GraderResult object that contains the Grader's state as well as a set of UIAtoms representing the query to the student. A UIAtom is the means by which a Grader communicates with students through the IHS; the IHS may display them in different ways (such as on a website, on a mobile device, as plain text, etc.), but this process is abstracted from the Grader.

The grade method is called by the IHS after the student supplies a response to a query. The Grader is again passed a GraderInfo object, containing the same Parameter and random seed as before as well as the state that was returned when the current query was created. Depending on the Grader's implementation, the Parameter, the seed, the state, and the student's response, the Grader returns a GraderResult containing either a final score (ending the attempt) or a new query to be answered by the student. In the latter case, the GraderResult contains the UIAtoms representing the new query and the Grader's new state. The grade method will be called after every student response until the attempt ends and the Grader assigns a score.

The getSampleParameter method is passed a random seed and one of ten predefined difficulty levels by the IHS, and it returns a Parameter object that the IHS can use to create a new attempt from the grader. In providing ten different levels of difficulty, the IHS makes it simple for instructors to break assignments up into smaller problems. Starting students on the easiest levels builds their confidence before they are introduced to more difficult problems, and the instructor needs only to change one setting to create an entirely new problem. Of course, instructors can create Parameter objects themselves – or tweak the Parameter objects returned by getSampleParameter – if they desire finer control over the appearance of assigned problems.

The final two methods of the Grader interface provide minor additional functionality. The getSupportedLocales method returns the list of all locales the Grader implementation supports; Graders support internationalization so that conversations between Graders and students can take place in different languages. The getResponder method returns a new

instance of the Grader's Responder implementation that is set up to provide the solution to the Grader's current query (obtainable by the Grader via the state object passed to `getResponder`).

4. The Factoring Problem Generator from the Users' Perspectives

In this chapter, the Factoring Problem Generator (FPG) is introduced. A brief look at the motivation behind creating factoring problems is presented first, followed by a description of the kinds of problems produced by the FPG – in other words, the problems as seen by students. This chapter concludes with an overview of the parameters configurable by instructors to tweak the problems generated by the FPG.

4.1. Motivation Behind the FPG

In 2005, a survey was conducted to understand the difficulties faced by beginning computer science students, and the results are presented in (Lahtinen 2005). 559 students and 34 instructors participated in the survey across six universities and over ten days.

In the survey, participants were asked to rate different programming issues on a 1-5 scale according to how difficult they are to learn. Of the seven presented issues, students ranked “dividing functionality into procedures” the third most difficult to learn, with “designing a program to solve a certain task” and “finding bugs in my own program” ranking second and first, respectively. Instructors ranked “dividing functionality into procedures” as the most difficult issue to learn (Lahtinen 2005).

Participants were also asked in the survey to rank different learning techniques on a 1-5 scale according to how useful they are to help students learn computer science. Students ranked “working alone on programming coursework” as the most useful technique.

Instructors ranked “practical sessions” as the most useful technique, with “working alone on programming coursework” ranking second (Lahtinen 2005).

It is seen, then, that separating functionality into procedures is a substantial hurdle for beginning programmers to overcome. According to the beginning programmers themselves, the best way to overcome such a hurdle is to write programs in which that hurdle presents itself. The FPG creates exactly that kind of problem: the kind in which students work independently to factor out common functionality into individual procedures. It is the aim of the FPG to provide a means by which students may both be gently introduced to and intensely drilled in using functions to factor out repeated code in their programs.

4.2. FPG Problems from the Student’s Perspective

The kind of problem generated by the FPG is examined in this section from the student’s perspective. As discussed in the previous section, beginning computer science students often have difficulty in separating the functionality of their programs into different procedures, or functions. (Please note: the term *function* will be used in place of *procedure*, *method*, or any of the several other terms referring to a subroutine of a program, as *function* is the term used in the C programming language in which FPG problems are presented.) The FPG provides a means by which students can practice extracting blocks of common code into functions, utilizing function parameters and return values to account for slight differences in the code blocks.

To begin an FPG problem, the student is presented with a full program. The FPG currently presents and accepts programs written in the C programming language, although the FPG concept could be applied to any procedural language. The presented program contains exactly one function, the main function, and that function consists of a long series of statements without any calls to other functions. Contained within that series of statements are blocks of similar (though not identical) code. The student's objective is to identify those blocks of repeated code and extract them into one or more functions, replacing the blocks of code with function calls. This process is referred to as *factoring*. Because several calls to a single function replace entire series of statements, the length of the overall program decreases. To successfully solve an FPG problem, then, the student must meet two criteria:

1. The student's factored program must contain no more than a specified number of tokens. This number is specified by the FPG and is fewer than the number of tokens in the unfactored program.
2. The student's factored program must maintain exactly the same functionality as the unfactored program. For any given input, the student's factored program must produce the same output as the unfactored program.

An example of the initial unfactored program presented to the student by the FPG is shown in Figure 4-1. This particular problem is representative of the easier levels of difficulty supported by the FPG, though the length of the overall program has been reduced to simplify the example presented here.

```

#include <stdio.h>

int main() {
    int a, b, c, d, e, f;

    /* Initialize variables. */

    scanf(" %d %d", &b, &c);

    /* Execute code. */

    for (a = 0; a < 4; a = a + 1) {
        printf("%d %d\n", b, c);
    }
    c = 3 - b;

    for (f = 0; f < 12; f = f + 1) {
        printf("%d\n", c);
    }

    d = c - c / 9 % 5;
    e = f * (d / 7) + c;
    a = (e - (f + e)) % 6;

    c = 9 * a;

    for (f = 0; f < 12; f = f + 1) {
        printf("%d\n", 4);
    }

    d = 4 - 4 / 9 % 5;
    e = f * (d / 7) + 4;
    b = (e - (f + e)) % 6;

    /* Print variables to stdout. */

    printf("%d %d %d\n", a, b, c);

    return 0;
}

```

Figure 4-1: FPG Problem as Presented to a Student

In this program, there are two similar blocks of code that can be replaced with calls to a single function. Figure 4-2 compares this unfactored program with a factored solution (note that there may be more than one program that meets the aforementioned solution criteria).

<pre> #include <stdio.h> int main() { int a, b, c, d, e, f; /* Initialize variables. */ scanf(" %d %d", &b, &c); /* Execute code. */ for (a = 0; a < 4; a = a + 1) { printf("%d %d\n", b, c); } c = 3 - b; for (f = 0; f < 12; f = f + 1) { printf("%d\n", c); } d = c - c / 9 % 5; e = f * (d / 7) + c; a = (e - (f + e)) % 6; c = 9 * a; for (f = 0; f < 12; f = f + 1) { printf("%d\n", 4); } d = 4 - 4 / 9 % 5; e = f * (d / 7) + 4; b = (e - (f + e)) % 6; /* Print variables to stdout. */ printf("%d %d %d\n", a, b, c); return 0; } </pre>	<pre> #include <stdio.h> int funcA(int w) { int x, y, z; for (z = 0; z < 12; z = z + 1) { printf("%d\n", w); } x = w - w / 9 % 5; y = z * (x / 7) + w; return (y - (z + y)) % 6; } int main() { int a, b, c; /* Initialize variables. */ scanf(" %d %d", &b, &c); /* Execute code. */ for (a = 0; a < 4; a = a + 1) { printf("%d %d\n", b, c); } c = 3 - b; a = funcA(c); c = 9 * a; b = funcA(4); /* Print variables to stdout. */ printf("%d %d %d\n", a, b, c); return 0; } </pre>
--	---

Figure 4-2: Unfactored Program (left) Compared to its Factored Solution (right)

The initial unfactored program appears on the left in Figure 4-2. The two blocks of similar code are highlighted. The goal of the problem is to extract these blocks into a single function, and this requires the two blocks to be identical. Observe that everywhere the variable “c” appears in the first block, a “4” appears in the second block. If “c” in the first block and “4” in the second block are replaced with a common variable “w”, the two blocks will be identical with the exception of the last line in each block. The

common variable “w” will become a formal parameter in the new function; “c” can be passed as the actual parameter for “w” in the first call to the function and “4” can be passed in the second call.

On the last line, observe that the variable “a” is assigned to in the first block and the variable “b” is assigned to in the second block. Both “a” and “b” are read outside of the common blocks of code, so neither “a” nor “b” may be replaced with a local variable in the function (because a local variable would not be available for use outside of that function). Instead, the value assigned to “a” in the first block and to “b” in the second block should be returned so that they may be used outside of the function. The final assignment in each block is therefore replaced with a return statement, and the return values from the first and second calls from the main function are assigned to “a” and “b”, respectively.

Finally, notice that the variables “d”, “e”, and “f” are not read outside of the common blocks of code. These can be replaced with local variables “x”, “y”, and “z” in the function. The new function can now be written and the common code in the main function replaced with calls to the new function. Once this is done, the local variables in the main function – “d”, “e”, and “f” – are no longer used, so the declarations for them are removed. The factored solution to the unfactored problem is shown on the right in Figure 4-2. The new function and the function calls that replace the common code are highlighted.

As previously mentioned, this example is representative of the easier difficult levels supported by the FPG. At harder levels, the factored solutions may contain

- More than one function other than main
- Functions other than main that contain calls to other functions
- Floating-point data types
- Arrays
- Pointers
- Functions with a greater number of formal parameters
- Function calls with longer expressions passed as actual parameters
- A wider variety of operators including bitwise operators

Additionally, it was noted earlier that this example was shortened to simplify the example. Had this been an actual FPG problem, the common code would have been repeated three or four times such that factoring it into a single function would reduce code size by a greater amount.

4.3. Adjusting FPG Problems as an Instructor

As was discussed earlier, one of the primary goals of the IHS as well as the FPG is to allow complex programming concepts to be broken down into smaller pieces, making them easier for beginning computer science students to grasp and keeping them from draining students' morale. To this end, the FPG allows the instructor to configure several different aspects of the FPG's problems such that students can begin with easier problems before moving slowly but surely to more and more difficult ones. In this section, the FPG configuration options instructors have at their disposal are presented.

An instructor wanting to adjust the problems created and graded by the FPG does so by setting the desired values in the FPG's Parameter object. That object is then passed both to the `initQuery` and `grade` methods of the FPG's Grader class (see Section 3.2). The values that may be set in the Parameter object are summarized in Table 4-1.

Table 4-1: Summary of Configurable Values in an FPG Parameter

Value Name	Description
<code>numFuncs</code>	The number of functions (including main) that will appear in the factored solution.
<code>numCallLevels</code>	The depth of function calls that will appear in the factored solution.
<code>numCallsToEachFunc</code>	The number of times each function is called in the factored solution.
<code>numMainStmts</code>	The approximate number of statements that will appear in the main function in the factored solution.
<code>numMidFuncStmts</code>	The approximate number of statements that will appear in each function other than main that calls other functions.
<code>numLeafFuncStmts</code>	The approximate number of statements that will appear in each function other than main that does not call other functions.
<code>usableOperators</code>	The set of operators that may be used throughout the program and the frequencies at which they occur.
<code>usableStmts</code>	The set of statement types that may be used throughout the program and the frequencies at which they occur. Indirectly controls whether or not arrays and/or pointers appear in the program.
<code>useDoubles</code>	Whether or not the program may contain variables of the double floating-point data type.
<code>useArraysAsParams</code>	Whether or not arrays will appear as formal parameters in the factored solution (not applicable if the program does not contain arrays).
<code>usePointersAsParams</code>	Whether or not pointers will appear as formal parameters in the factored solution (not applicable if the program does not contain pointers).
<code>maxSimpleParams</code>	The maximum number of formal parameters that may be declared for any function in the factored solution (array and pointer parameters may exceed this limit).
<code>maxOpsInParamExpr</code>	The maximum number of operators that may appear in expressions passed as actual parameters in the factored solution.
<code>numProblems</code>	The number of programs the student must successfully factor in order to pass an attempt.
<code>numTriesPerProblem</code>	The number of responses the student may submit attempting to factor a single program.

An instructor has the option of either creating an FPG Parameter object manually and setting each of its values or using the `getSampleParameter` method of the FPG's Grader class to get a default Parameter and modifying zero or more of the values as desired. In either of these scenarios, it is possible that the instructor sets an invalid value for one or more of these parameter values. In that case, the FPG ignores the instructor's value and instead uses a preprogrammed default value.

The `numFuncs`, `numCallLevels`, and `numCallsToEachFunc` values control the function call graph of the solution program. The `numFuncs` value specifies the total number of functions (including main) that will appear in the factored solution program. A higher value results in a greater number of functions, and this generally results in a longer unfactored program presented to the student. The `numCallLevels` value controls the depth of function calls within the program. If, for example, `numCallLevels` was set to two, the factored program would contain two functions A and B such that main calls A, A calls B, and B does not call any functions. Furthermore, it would be guaranteed that there would be no functions A, B, and C such that main calls A, A calls B, and B calls C. A higher `numCallLevels` value results in more complex and often lengthier code for the student to factor. Finally, the `numCallsToEachFunc` value controls the number of calls to each function that appear somewhere in the factored solution. A higher value makes repeated blocks easier to spot but results in lengthier code.

The `numMainStmts`, `numMidFuncStmts`, and `numLeafFuncStmts` values each control the approximate number of statements that should appear in the factored solution program. The `numMainStmts` value applies to the main function, the `numMidFuncStmts` value

applies to functions other than main that call other functions, and the numLeafFuncStmts applies to functions other than main that do not call other functions. Instructors can only set the approximate number (as opposed to the exact number) of statements in functions because the FPG must retain the freedom to add or remove statements as it sees fit in order to ensure sensible programs that do not contain useless code. A more detailed explanation is given in the next chapter. The effect that each of these three values has on overall problem difficulty is unclear; a greater number of statements results in more code to examine but may make repeated blocks of code easier to spot.

The usableOperators and usableStmts values give the instructor fine control over the appearance of expressions and statements within the FPG program. The usableOperators value allows the instructor to choose not only the set of operators that may appear in the program but the approximate frequency at which they appear as well. Similarly, the usableStmts value allows the instructor to choose allowable types of statements as well as the frequency at which they appear. The FPG utilizes eight different types of statements, and they are listed in Table 4-2.

Table 4-2: Types of Statements Used in Programs Generated by the FPG

Name	Description
ASSIGN	A statement that assigns into a simple variable.
IF_ELSE	An if block with optional else if and else blocks.
FOR	A for loop.
WHILE	A while loop.
ARRAY	A loop that iterates over the elements of an array, performing an operation.
ASSIGN_PTR	A statement that changes where a pointer points, i.e. a statement of the form <code>ptr = &var;</code> or <code>ptrA = ptrB;</code>
DEREF_PTR	A statement that assigns into the target of a pointer, i.e. a statement of the form <code>*ptr = expr;</code>
PRINTF	A call to the <code>stdio.h</code> <code>printf</code> function, printing one or more values to standard output.

The `usableStmts` value also controls whether or not pointers and arrays appear in the FPG program. If the set of allowable statements contains the `ARRAY` statement type, then one or more arrays will be declared in the program. Similarly, if either the `ASSIGN_PTR` or `DEREF_PTR` types appear in the set of allowable statements, then one or more pointers will be declared in the program. It is an invalid value if one of the `ASSIGN_PTR` and `DEREF_PTR` types appears in the `usableStmts` set without the other, and in that case the FPG will add the other into the set.

The `useDoubles`, `useArraysAsParams`, and `usePointersAsParams` values have control over the types of variables that may be declared in the FPG program, with the latter two applying specifically to formal parameters of functions in the factored solution. If the instructor allows doubles to be used, then variables of the double data type will be declared both as local variables and as formal parameters, and they will be used throughout the program. If the instructor allows arrays or pointers to be used as

parameters, then array formal parameters and pointer formal parameters, respectively, will be declared. These two values only have an effect if arrays or pointers appear in the program, and this is controlled by the `usableStmts` value as explained above.

The `maxSimpleParams` and `maxOpsInParamExpr` values provide finer control over how parameters to functions are utilized in the factored solutions to FPG problems. The `maxSimpleParams` value controls the maximum number of formal parameters that will be declared for any function in the factored solution, excluding array and pointer parameters (as these are controlled by the `useArraysAsParams` and `usePointersAsParams` values, respectively). The `maxOpsInParamExpr` value controls the maximum number of operators that may appear in expressions passed as actual parameters in the factored solution. This value greatly affects the difficulty of FPG problems when combined with the `numCallLevels` value. With a greater number of operators in expressions (and therefore greater lengths of expressions), it becomes more difficult to factor parameters out of expressions hard-coded into a function. Consider the example of Figure 4-3.

<pre>int main () { ... y = x + 4; ... y = c + 4; ... }</pre>	<pre>int main () { ... y = x + 4; ... y = 3 * c - d + 4; ... }</pre>
--	--

Figure 4-3: Effect of the `maxOpsInParamExpr` Parameter Value on Difficulty

A student may observe in the program on the left that the two shown statements are similar. These statements could be factored into a function that utilizes a parameter, and

the calls from main replacing the repeated code would pass “x” as the actual parameter in the first case and “c” as the actual parameter in the second case. In each case, no operators appear in the expression passed as an actual parameter. A similar scenario is illustrated in the program on the right, but the `maxOpsInParamExpr` is increased from zero to two. Here, the shown statements could be factored into a function that utilizes a parameter, passing “x” as the actual parameter in the first call and “3 * c - d” as the actual parameter in the second call. The two statements in the unfactored program, however, look quite different, and a student may fail to notice the similarity. This issue becomes even more pronounced as the depth of function calls increases. Therefore, great care must be used when adjusting this value, as even a slight increase in the allowed number of operators in parameters can cause a dramatic increase in overall problem difficulty.

Finally, the `numProblems` and `numTriesPerProblem` values control how students are graded through a single attempt. The `numProblems` value dictates the number of programs a student must successfully factor in one IHS attempt in order to pass that attempt. Because the FPG generates random programs, some may be easier to complete than others. The instructor can set the `numProblems` value to two or more in order to protect against fluke programs that are very easy to factor. The `numTriesPerProblem` controls the number of responses a student may submit for a single program before failing the entire attempt. If the student’s response fails to compile, that response does not count against him. But if the student’s response compiles and either fails to match the initial unfactored program’s output or contains too many tokens, the student moves on to the next response or, if no responses are left, fails the attempt.

5. Implementation of the Factoring Problem Generator

This chapter examines the implementation of the Factoring Problem Generator (FPG), including the major hurdles that had to be overcome during design and development. The process of creating an initial problem to be given to the student is presented as well as the process of grading the student's response.

The FPG is coded in Java so that it may fit into the IHS framework. When it receives a request from the IHS to generate a new problem, the FPG actually creates the factored solution program first. This program contains two or more functions, and it is the program given by the FPG when it responds to its own queries. To obtain the unfactored program that is given to the student as the initial problem, the FPG simply inlines all of the functions into one function, main. In other words, the FPG performs exactly the opposite operation on the code that the student performs as the problem is solved.

This chapter begins with an examination of the process through which the FPG creates function headers and the function call graph of the factored program. It continues with an explanation of how the FPG generates code to fill those functions. Finally, it presents the FPG's process for creating the test cases used to verify that the student's factored solution maintains the same functionality as the initial unfactored program.

5.1. Creating Function Headers and the Function Call Graph

This section examines the process through which the FPG creates functions and a function call graph for its factored solution program. This process is the starting point for the FPG when a new problem is requested by the IHS, and it is controlled largely by the numFuncs, numCallLevels, and numCallsToEachFunc values of the FPG's Parameter.

5.1.1. Creating the Function Call Graph

To begin the process of creating a program, the FPG creates a function call graph. This call graph controls what functions are contained in the program as well as what calls each function will make somewhere within its body. In creating the call graph, the FPG must ensure that the graph contains no cycles (e.g. A calls B and B calls A, or the recursive case of A calls A). Otherwise, the inlining of the factored solution into the initial unfactored program will fail. To ensure that the call graph contains no cycles, each function is assigned to a *call level*. In an FPG program, a function may only call functions in a higher call level than its own. Therefore, if A can call B, then B must be in a higher call level than A and thus B cannot call A.

An illustration of the call graph creation process for a particular example program is shown in Figure 5-1. In this example, the numFunctions value is set to seven, the numCallLevels value is set to three, and the numCallsToEachFunction is set to two. Each box represents a function, and each arrow represents one function call appearing somewhere in the calling function's body. Arrows point from the calling function to the called function. The functions are arranged in columns by call level.

(a) Initial Chain of Function Calls

(b) Remaining Functions Added to Random Call Levels

(c) Remaining Function Calls Added

Figure 5-1: Process of Creating the Function Call Graph

To begin, the FPG creates the main function and assigns it a call level of zero. It then creates one function for each remaining call level. The FPG adds one call from each function to the function in the next highest call level as shown in Figure 5-1 (a). This initial call chain guarantees that the final FPG program contains at least one chain of function calls that is as deep as specified by the numCallLevels value.

The FPG next creates any remaining functions as specified by the numFuncs value. These functions are assigned to random call levels with the condition that no function other than main may be in call level zero. See Figure 5-1 (b).

Finally, the FPG adds calls randomly to each function from functions in lower call levels. At the end of this process, each function aside from main is called the number of times specified by the numCallsToEachFunc value as shown in Figure 5-1 (c).

5.1.2. Creating Variables for Each Function

Once the call graph is created, the FPG creates the variables for each function, including both formal parameters and local variables. These variables are more or less randomly chosen, but the FPG must ensure that calling functions have the ability to pass an actual parameter for each formal parameter in the called function. This is rarely an issue for certain kinds of parameters, but others tend to present a greater problem.

At this point, it is appropriate to introduce the classes of variable supported in programs generated by the FPG. In an FPG program, a variable can be classified both by its type and by its kind. The term *type* refers to the elemental data type of a variable, such as int

or char. The FPG supports two C data types: int and double. The term *kind* is used within the FPG to differentiate between arrays, pointers, and simple variables, each of which is supported by the FPG. The terms *array* and *pointer* are used within the FPG with the same meaning as they are used within the C language, and the term *simple variable* refers to a variable of one of the supported types that is not an array or pointer. The FPG does not support C structures at this point, though they may be added in the future. The variable classes that are supported by the FPG are shown in Table 5-1 along with sample variable declarations as they might appear in an FPG program.

Table 5-1: Classes of Variables Supported by the FPG

Type	Kind	Sample Declaration
int	simple	int x;
int	array	int x[5];
int	pointer	int *x;
double	simple	double x;
double	array	double x[7];
double	pointer	double *x;

The FPG adds local simple variables randomly to each function, and it adds local arrays and pointers to main according to the settings in the FPG's Parameter. Local arrays and pointers are not added to functions other than main because they tend to contain considerably less code and it is difficult to make use of them with the limited number of statements. Instead, arrays and pointers are used as formal parameters to functions.

The selection of formal parameters for each function is a slightly more difficult problem. For simple parameters, a caller can always pass a literal constant if no appropriate variables are available. But literals cannot be used as actual parameters for arrays or pointers, and the calling function must have access to an appropriate variable to pass. For example, if a function has an int array as a formal parameter, functions that call that function must have either a local int array or an int array as a formal parameter. The FPG avoids this problem by only allowing functions at the first call level (i.e. functions called only by main) to have formal parameters that are of the array or pointer kind. The main function always contains arrays and pointers if they are used anywhere within the program, so main is guaranteed to have an appropriate variable to pass to any function call.

5.2. *Filling in Functions with Meaningful Code*

Once the function headers and the function call graph are created, the FPG fills in each function with code. The FPG cannot, however, simply generate random statements without regard to context. Consider the code excerpt in Figure 5-2.

```
(1)  int x, y, z;  
(2)  
(3)  scanf("%d", &z);  
(4)  y = z + 3;  
(5)  x = 7 + z % 10;  
(6)  y = x - z;  
(7)  printf("%d %d %d", x, y, z);
```

Figure 5-2: Example of Meaningless Code

A value is read from the user and stored in the variable “z” on line 3 of this excerpt. Values are assigned to “y” and “x” in the following two lines, and the expressions used to calculate those values both contain “z” and therefore depend on the value entered by the user. On line 6, however, a new value is calculated and assigned to “y”. The value stored in “y” on line 4 was not used before being overwritten, and line 4 can therefore be removed from the program without affecting its output in any way.

If this phenomenon were to occur in an FPG program, it could provide an opportunity for the student to supply a response that the FPG considers correct but that does not properly factor out all repeated code. The FPG evaluates the amount of code factoring by counting tokens in the student’s response, so if the student can significantly reduce the token count by deleting meaningless statements then the intended FPG factoring process may be circumvented.

In order to avoid creating meaningless statements, the FPG maintains two lists of variables as it fills each function with code. The *must assign list (MAL)* contains variables to which an assignment is needed. A simple variable in the MAL does not contain a usable value, most often because it has not been initialized. Unlike simple variables, the FPG guarantees that arrays always contain usable values and may always be safely read. An array in the MAL, then, indicates an array that needs to be written to in order to make its existence meaningful. This occurs when a function has an array as a formal parameter; the FPG requires that the array be written to in the body of the function so that calls to functions modify any arrays passed as actual parameters. Finally, a pointer in the MAL indicates a pointer whose target should be written to before the end of

the current function and before the pointer is made to point to a different target. Like arrays, this happens when a function has a pointer as a formal parameter; the FPG forces the pointer to be written to somewhere in the function so that the pointer is an “out” parameter. All pointers, however, are guaranteed by the FPG to always point to a variable containing a valid value. Note that it is an error to read from a simple variable in the MAL, but it is not an error to read from arrays and pointers in the MAL.

The other list maintained by the FPG while creating statements is the *must read list* (*MRL*). The MRL contains variables that contain values that have not yet been read and therefore should not be overwritten. Because of some simplifications, the FPG needs to use this list only for simple variables and not for arrays or pointers. Assignments to array elements always contain the array element on the right-hand side of the assignment, so each element’s value is used immediately before it is overwritten and hence arrays do not need to be kept track of in the MRL. Pointers are guaranteed by the MAL to have their targets written to before they are made to point to a different variable, and thus their existence is meaningful even if their targets are never read from. Pointers, then, also do not need to be kept track of in the MRL.

5.2.1. Creating Meaningful Series of Non-Branching Statements

Now that the MAL and the MRL have been explained, an example is presented to illustrate the process through which the FPG generates statements. If/else blocks and loops present a higher degree of complexity, so a second example using such statements is presented in the next subsection.

Consider the code shown in Figure 5-3 illustrating a specific example of the process the FPG goes through to create statements using the MAL and the MRL. At the start of this process, the function header and the local variable declarations shown on lines 1 and 3 would exist, but the statements on lines 8 through 28 would not. To the right of each statement appear the MAL and the MRL immediately after the creation of the statement.

(1)	int funcA(int a, int b, int *ptr) {		
(2)			
(3)	int x, y;	<u>MAL</u> x, y, ptr	<u>MRL</u> a, b
(4)			
(5)			
(6)			
(7)			
(8)	x = a * 4;	<u>MAL</u> y, ptr	<u>MRL</u> b, x
(9)			
(10)			
(11)			
(12)			
(13)	printf("%d %d", x, *ptr);	<u>MAL</u> y, ptr	<u>MRL</u> b
(14)			
(15)			
(16)			
(17)			
(18)	y = 2 - x / 3;	<u>MAL</u> ptr	<u>MRL</u> b, y
(19)			
(20)			
(21)			
(22)			
(23)	*ptr = y + x;	<u>MAL</u>	<u>MRL</u> b
(24)			
(25)			
(26)			
(27)			
(28)	return x / b * 2;	<u>MAL</u>	<u>MRL</u>
(29)			
(30)			
(31)	}		

Figure 5-3: Process of Creating Non-Branching Code

At the end of line 3, local variables have been declared but no statements have been executed. The parameters “a” and “b” appear in the MRL because the values passed to them when the function is called should be read somewhere in the function. The local

simples “x” and “y” appear in the MAL because they are uninitialized and cannot be read. The pointer parameter “ptr” also appears in the MAL because its target must be assigned to in the function body. Recall, however, that pointers in the FPG are always initialized to point to initialized variables, so a pointer can always be read from. At this point, the FPG begins creating statements. At the end of the process, it is desirable that no variables appear in either the MAL or the MRL indicating that all variables were initialized and read from.

On line 8, the FPG decides to create an assignment statement. The FPG can choose to assign to any variable that is not on the MRL, but it gives priority to variables on the MAL because an empty list is desired. It chooses the variable “x” to assign to and proceeds to create an expression for the right-hand side of the assignment. In such an expression, the FPG can use constants as well as any variables that do not appear on the MAL, but it gives priority to variables on the MRL because again an empty list is desired. Having been assigned a value, “x” is removed from the MAL and inserted into the MRL. The parameter “a” is removed from the MRL because of its use in the assigned expression. Note that removing “x” from the MAL prior to creating the assigned expression would allow “x” to be used within that expression, and this is an error condition because “x” would remain uninitialized at the time the expression is evaluated.

The FPG next decides to create a call to the printf function. Such a call never assigns to variables and thus can only affect the MRL. The FPG decides to print the values stored in “x” and in the target of “ptr”. Because “x” is read here, it is removed from the MRL.

The pointer remains on the MAL because it still has not been assigned to. Again, note that a pointer's target can always be read even if the pointer appears in the MAL.

The FPG creates another assignment statement on line 18, choosing the variable “y” from the MAL to assign to. In the expression created on the right-hand side of the assignment, the variable “x” is used. Even though “x” does not appear in the MRL, it does not appear in the MAL and is therefore safe to read. Because it has been assigned a value, “y” is moved from the MAL to the MRL.

At this point, the FPG decides to begin the process of ending the function. The FPG must empty the MAL in order to do so, so it decides to assign into the target of “ptr” on line 23. It uses “y” in the assigned expression so that it may be removed from the MRL, and “x” can again be used because it remains unseen in the MAL. With the MAL empty, the FPG can conclude the function. It uses every variable appearing in the MRL in the expression that it creates for the return statement on line 28, and each of these can be removed from the list. If it had been the case that many variables remained on the MRL, the FPG could have decided to insert one last call to the printf function to lessen the number of variables appearing on the list. This was not the case, however, and the FPG is able to finish the function with an empty MAL and an empty MRL.

5.2.2. Creating Meaningful Series of Statements with Branches

As soon as branching statements are introduced into the mix, the maintenance of the MAL and the MRL becomes more complicated. Consider the example shown in Figure

5-4. The MAL and MRL are shown three times: once after the declarations but before any code is written, once immediately after the creation of the if/else block, and once immediately after the for loop.

Figure 5-4: Process of Creating Branching Code

After the declaration on line 3, the uninitialized local appears in the MAL. The array parameter “arr” also appears in the MAL in order to guarantee that it will be modified somewhere in this function. The simple parameter “a” appears in the MRL to force its use within the function body.

The FPG decides to create an if/else block as the first statement in the function. The parameter “a” is read when the if’s condition is evaluated, so it is removed from the MRL. When the FPG enters the body of an if, else, or loop, it makes a copy of the MAL and MRL so that when the FPG exits the block the MAL and MRL can be returned to their original states. In this example, the FPG assigns to the local variable “x” within the

body of the if, and “x” is subsequently read by a printf call. As a result of these two statements, the “x” is moved from the MAL to the MRL and then off of the MRL (this is not shown in the figure). However, the body of the if ends and the original MAL and MRL are restored.

The FPG then moves on to the body of the else, again making copies of the MAL and MRL. It decides to create a printf call on line 11, and because “x” is on the MAL it cannot be used in the statement. The FPG exits the else body and must update the original MAL and MRL. The array “arr” was not used anywhere in the if/else statement, so it remains on the MAL. The local variable “x” was used in the if body and was removed from both the MAL and the MRL. However, “x” was not used in the else body, so it is entirely possible that “x” will remain uninitialized at this point. Therefore, “x” must be kept in the MAL at the conclusion of the if/else statement.

In general, the FPG may compare several MALs and MRLs at the conclusion of a branching statement like this one. It follows these three rules to consolidate the lists into a single MAL and MRL:

1. If a variable appears in any MAL, place it in the MAL.
2. If a variable does not appear in any MAL but appears in one or more MRLs, place it in the MRL.
3. If a variable does not appear in any of the lists, do not place it in the MAL or the MRL.

Consider the significance of Rules 1 and 2 giving precedence to variables in the MAL. If a variable appears on the MAL from one branch and on the MRL from another, that variable will be placed into the MAL following the branching statement. This will cause that variable to be assigned to, erasing the value that was previously assigned in the branching statement (which would have occurred in the block that had the variable in the MRL). In this case, the FPG may add a printf call using variables that were added to the MRL in the branch such that their values are certain to be used. In fact, the printf on line 8 in Figure 5-4 serves this purpose.

Finally, the FPG adds an array statement, utilizing a for loop. The local variable “x” is used as the count variable, so it is both initialized and read at the conclusion of the loop and can be removed from both the MAL and the MRL. The array is modified on line 16, so it can also be removed from the MAL. Recall that arrays only appear in the MAL and are not kept track of on the MRL. At this point, the FPG decides to conclude the function. Because the MAL and MRL are both empty and the function does not return a value, the FPG does not need to add any more statements.

5.2.3. Creating Meaningful Statements Within Loops

When creating statements inside a loop body, the FPG must do more than utilize the MAL and MRL to guard against useless code. Consider the loop shown in Figure 5-5.

```
(1)  while (x < 5) {  
(2)      y = 12;  
(3)      x = x + 1;  
(4)  }
```

Figure 5-5: Loop Containing Meaningless Code

This loop will be executed for certain values of “x”, and it will iterate any number of times depending on the exact value. However, the assignment on line 2 has the same effect on the overall program if it is run one time or if it is run several times. It is preferable that the statements in a loop body be written in such a way that the exact number of times they are executed matters to the overall program. The FPG accomplishes this by creating assignment statements in loops where the variable being assigned to is also used on the right-hand side of the assignment. This ensures that the value assigned to a variable in an iteration of a loop depends on the value assigned in the previous iteration of that loop.

Finally, the FPG must ensure that no infinite loops are created. In an FPG program, the execution of a loop is always controlled by a count variable, such as “x” in Figure 5-5. When creating statements for a loop, the FPG never assigns to the count variable except for the statements specifically designed to control the loop (like the assignment on line 3 in Figure 5-5). However, it is safe for the FPG to read from the count variable anywhere within the loop.

5.3. Inlining the Solution into the Problem

Once the FPG has created all necessary functions and filled them with meaningful code, it inlines those functions into the single main function. The inlined program is then presented to the students as the initial unfactored program.

5.3.1. The Inlining Process

The inlining process is a well-studied problem, but it has been developed principally to optimize code. The FPG is unconcerned with optimization and inlines instead for a different purpose. Consider Figure 5-6, where a simple approach to inlining is shown.

<pre>(1) #include <stdio.h> (2) (3) int funcA(int a, int b) { (4) int z; (5) (6) z = a + 9; (7) (8) return z - a / b; (9) } (10) (11) int main() { (12) int x, y; (13) scanf("%d", &y); (14) (15) if (y < 2) { (16) x = y; (17) } (18) else { (19) x = 8; (20) } (21) (22) y = funcA(x, 3); (23) (24) printf("%d", y); (25) (26) x = funcA(4 + x, y); (27) (28) printf("%d %d", x, y); (29) return 0; (30) }</pre>	<pre>(1) #include <stdio.h> (2) (3) int main() { (4) int x, y, z, a, b; (5) scanf("%d", &y); (6) (7) if (y < 2) { (8) x = y; (9) } (10) else { (11) x = 8; (12) } (13) (14) a = x; (15) b = 3; (16) z = a + 9; (17) y = z - a / b; (18) (19) printf("%d", y); (20) (21) a = 4 + x; (22) b = y; (23) z = a + 9; (24) x = z - a / b; (25) (26) printf("%d %d", x, y); (27) return 0; (28) }</pre>
--	---

Figure 5-6: The Simple Inlining Process

In this figure, the program generated by the FPG is shown on the left and the inlined program is shown on the right. To inline the program, each call to “funcA” is replaced with the code in the function’s body. Immediately preceding that code is a series of assignment statements, highlighted on the right side of Figure 5-6. These statements initialize the formal parameters used in the function body. Because this series of assignment statements precedes every call to the function, the repeated code becomes very easy to spot. Furthermore, the statements make it simple to determine what the formal parameters should be and what actual parameters should be passed to each function call in the factored solution.

To avoid these issues, the FPG takes a different approach to inlining. The FPG does not set the values for formal parameters and then copy the code in the inlined function’s body directly as in the above example. Instead, the FPG enforces the rule that formal parameters may be read from but not written to. In other words, the FPG treats formal parameters as immutable, allowing the FPG to directly substitute the actual parameters for a particular call of a function directly into that function’s body. This has the advantages that the series of assignment statements initializing the formal parameters are not needed and that the blocks of inlined code look similar but *not* identical. The program of Figure 5-6 is shown inlined again in Figure 5-7, this time using the FPG’s inlining process.

<pre> (1) #include <stdio.h> (2) (3) int funcA(int a, int b) { (4) int z; (5) (6) z = a + 9; (7) (8) return z - a / b; (9) } (10) (11) int main() { (12) int x, y; (13) scanf("%d", &y); (14) (15) if (y < 2) { (16) x = y; (17) } (18) else { (19) x = 8; (20) } (21) (22) y = funcA(x, 3); (23) (24) printf("%d", y); (25) (26) x = funcA(4 + x, y); (27) (28) printf("%d %d", x, y); (29) return 0; (30) }</pre>	<pre> (1) #include <stdio.h> (2) (3) int main() { (4) int x, y, z; (5) scanf("%d", &y); (6) (7) if (y < 2) { (8) x = y; (9) } (10) else { (11) x = 8; (12) } (13) (14) z = x + 9; (15) y = z - x / 3; (16) (17) printf("%d", y); (18) (19) z = 4 + x + 9; (20) x = z - (4 + x) / y; (21) (22) printf("%d %d", x, y); (23) return 0; (24) }</pre>
---	--

Figure 5-7: The FPG Inlining Process

5.3.2. *Choosing Actual Parameters for Function Calls*

When the FPG inlines a function as above with no arrays or pointers as formal parameters, the process is simple. But the introduction of arrays and pointers causes a new problem: the parameters are not quite immutable as the FPG assumes. In fact, the FPG actually guarantees that code is created to modify any array formal parameters and to assign to any pointer formal parameters. Because the actual parameter expressions are directly substituted into code, the values to which those expressions evaluate cannot change. If they do, the inlined program will behave differently than the factored program.

The FPG, then, must choose actual parameters in such a way that arrays and pointers can be modified as required but that simple variables retain constant values throughout the execution of the function. The algorithm used by the FPG to select actual parameters for a particular function call is shown in Figure 5-8.

```

( 1) usableList ← ∅
( 2) unusableList ← ∅
( 3)
( 4) FOR (each array formal)
( 5)     select an array to pass as actual
( 6)     add the selected array to unusableList
( 7)
( 8) FOR (each pointer formal)
( 9)     select either a pointer or a simple to pass as actual
(10)     add the selected variable to unusableList
(11)
(12) FOR (each variable v in caller's parameters and locals)
(13)     IF
(14)         v is a simple
(15)         AND v is not in unusableList
(16)         AND no pointer of the same type as v is in unusableList
(17)     THEN
(18)         add v to usableList
(19)     ELSE IF
(20)         v is a pointer
(21)         AND no simple of the same type as v is in unusableList
(22)         AND no pointer of the same type as v is in unusableList
(23)     THEN
(24)         add v to usableList
(25)     ELSE IF
(26)         v is an array
(27)         AND v is not in unusableList
(28)     THEN
(29)         add v to usableList
(30)
(31) FOR (each simple formal)
(32)     create an expression to pass as actual using only variables
        in usableList and constants

```

Figure 5-8: The Actual Parameter Selection Algorithm

To begin the actual parameter selection process, the FPG selects arrays to pass to the array formal parameters. It then moves on to pointer formal parameters. Each pointer must be initialized with an address, and the FPG either uses the address stored in another

pointer or passes the address of a specific simple variable using C's "address of" ampersand operator. Each variable used to initialize either an array or a pointer is placed on the "unusables" list.

The FPG then constructs the "usables" list from the "unusables" list. This new list will contain all variables that are safe to use in expressions that initialize simple formal parameters. In other words, the "usables" list contains all variables that are not able to be modified given the selection of array and pointer actual parameters. All formal parameters and local variables of the calling function are candidates to be placed on this list, so the FPG iterates through each of those to build the "usables" list.

A simple variable in the calling function may have its value changed during the execution of the called function only if a pointer formal parameter contains its address. This can happen in two ways: either its address was used to initialize a pointer formal parameter, or its address was contained in a pointer that was used to initialize a pointer formal parameter. The first of these is checked for on line 15 of Figure 5-8. The second is guaranteed not to be the case only if no pointers of the same type were used to initialize a pointer formal parameter (because any of those pointers could have contained the address of the simple variable in question), and this is checked for on line 16. Any simple variable that cannot have its value changed is added to the "usables" list.

Similarly, pointers may only be used in simple actual parameter expressions if the variables to which they point are guaranteed not to change. A pointer's target in the calling function can have its value changed in two ways: either the target was used to

initialize a pointer formal parameter, or a pointer containing the address of the target was used to initialize a pointer formal parameter. Because pointers may point to any variable of the same type, the FPG cannot be sure of where a pointer will point at every given time. Therefore, if any simple variable of the same type was used to initialize a pointer formal parameter (checked for on line 21 in Figure 5-8), the pointer in question may be pointing to that variable and therefore the target of the pointer in question may have its value changed. Furthermore, if any pointer of the same type was used to initialize a pointer formal parameter (checked for on line 22), it may point to the same variable as the pointer in question and therefore the target of the pointer in question may have its value changed.

Finally, because the FPG disallows pointers pointing to array elements for simplicity's sake, the FPG does not need to worry about pointers when deciding whether to use elements in arrays when creating actual parameter expressions. Instead, whether or not elements in an array may be used depends only on whether or not that array was used to initialize an array formal parameter.

After the “usables” list is created, the FPG uses variables within that list as well as constants to create the expressions passed as actual parameters in the function call. Variables that do not appear in the “usables” list will not appear in any of these expressions.

5.4. *Creating Test Cases*

The final task that the FPG must complete before presenting the created problem to the student is creating test cases that will be used to verify that the student's response maintains the same functionality as the initial unfactored program. The term *test case* here refers to the input that will be fed to an executing program. The goal when creating the FPG was to generate a set of test cases for each problem that would result in complete code coverage. In other words, for each branch in the code, there would be at least one test case in the set that causes it to execute. The automated creation of test cases is not a new problem, nor is it an unexplored one. But it is guaranteed that for some programs it is impossible to achieve complete code coverage with any set of test cases. For example, a program might contain an if block whose condition always evaluates to false. A system that tries to achieve complete code coverage will fail when creating test cases for such a program.

Unlike a test case generator, however, the FPG does not need to be able to generate test cases for any arbitrary program. Instead, it only generates test cases for programs that it creates itself. The FPG can enforce any rules necessary in designing programs to ensure that it is possible to achieve complete code coverage from the test cases. In this section, we examine three methods of generating test cases that were considered for use in the FPG. Two of these methods were deemed too complex for practical use, and they are presented here only in brief. The third method, the simplest of the three and the one used to implement the FPG, is presented last.

5.4.1. Backward Code Computation

In the first method considered for generating test cases, the FPG would first create the program as described in the previous sections of this chapter. The FPG would then step backwards through the code from each branch to the start of the program, calculating the conditions that would need to be met to cause the branch to execute and selecting a program input based on those conditions. An example illustrating this process is shown in Figure 5-9.

Figure 5-9: Finding Starting Conditions through Backward Code Computation

Here, the goal is to find the conditions that must be met to cause a thread of execution starting at Point A on line 1 to execute the statements beginning at Point B on line 14. The FPG would start at line 14 and trace the code backward, maintaining a set of conditions as it went. Maintaining the set of conditions is straightforward for non-branching statements like assignments, but if/else blocks, loops, and function calls cause the process to become much more complicated. Furthermore, the set of conditions that

are calculated for programs of significant size are often so complex that it becomes difficult to find a set of inputs that meets the conditions.

Finally, it should be noted that because the FPG would create the program first and the test cases second in using this method, it would not be guaranteed that every branch in a program is reachable. Indeed, it would be possible for the FPG to calculate a set of conditions that could not possibly be met.

5.4.2. Function Building Blocks

A second method considered for generating test cases was through *function building blocks*. In this method, the FPG would take an entirely different approach not only to generate test cases, but to generate programs as well. Instead of generating programs statement by statement, the FPG would provide an interface through which instructors could define their own functions – the “building blocks” of a program. Each function would have “plug-in points” – statements in the function body that could be replaced with calls to other functions. To assemble a program, the FPG would simply select a subset of the available function building blocks and link them together by replacing the plug-in points in some functions with calls to the other functions.

The building block interface would also allow instructors to specify conditions for test cases for each block. As the FPG assembles the blocks into a program, it could execute a backward computation to determine test cases for the entire program, similar to the process used in the backward code computation method described above. Doing this

could produce test cases that result in complete code coverage, and the computations would only need to be done at the function level rather than at the statement level.

This method has the advantage of allowing instructors to write their own code for use in FPG programs, giving them complete customization over the system. However, instructors could still create function building blocks that, in certain combinations, would prevent the FPG from achieving complete code coverage with its test cases. Furthermore, though this method is likely to be simpler than the backward code computation method, tracing the program in reverse can still result in highly complex conditions that test cases must meet. For this reason, the FPG does not use this approach.

5.4.3. Generating the Test Cases Before the Program

The backward code computation method and the function building block method both have one thing in common: the program code is generated first, and the test cases to exercise the program are generated second. The FPG, however, has the unique advantage of only needing to generate inputs for its own programs instead of for any arbitrary program, and this removes the necessity to generate test cases *in response* to a supplied program. The test cases can instead be generated in parallel with the program, or even *before* the program. And in fact, this is the method actually used by the FPG to create testing inputs for its programs: the FPG first creates a set of test cases, and then it generates the program such that for each branch in the code, there is at least one test case that causes the branch to execute.

The FPG generates the set of test cases for a program after the function call graph has been generated and the function parameters and locals have been declared, but before any of the functions have been filled with code. Every FPG program begins with one or more calls to the `scanf` function to read values from standard input, initializing a subset of the local variables in the main function. Each test case thus consists of values for certain variables that are input to the program at the beginning of the main function before any code executes.

Once the test cases have been created, the FPG proceeds to write code for each function. As it proceeds, it maintains a set of states for the program. A *state* contains the value for each variable in the program at a given time that result from the input of a particular test case. After the creation of each statement, the FPG “executes” that statement, updating the values in each state. Thus, as the FPG creates the program, it executes the code in parallel for each generated test case.

When the FPG decides to write a branching statement, it uses the information contained in the set of states to create branching conditions such that the branch will be taken for at least one of the test cases. This not only results in complete code coverage, but it also ensures that the FPG cannot write unreachable, meaningless code. Additionally, this method also helps guard against infinite loops because the FPG executes every statement it writes. An example of the process is shown in Figure 5-10. In the figure, each arrow points from a statement to the states resulting from the execution of that statement. Variables in states whose values have just changed are highlighted in red.

Figure 5-10: Maintaining States as the FPG Creates Statements

At the start of the process, the FPG has already created function headers (with parameters) and local variables. Additionally, the FPG has already selected a subset of local variables in main that will be initialized by calling the scanf function. For this subset, the FPG selects the values that will be passed to the program for each test case. In this simplified example, the FPG creates only two test cases and uses them to create the states of the program immediately after the initializations on lines 7 and 8. Observe that some variables (“y” in this case) are not initialized.

The FPG is now ready to begin writing code. It decides to create the assignment statement on line 12, following the process described in Section 5.2. Once the statement is created, the FPG updates the values in each state to reflect the execution of the statement.

The FPG next decides to create an if/else block. For the condition of the if, the FPG creates a random expression, in this case “ $y \geq x$ ”. Before it can use this expression, however, it must ensure that the expression evaluates to true in at least one state. The FPG finds that State A results in a true value, so the expression is acceptable. Had the FPG been unable to find a state that caused the expression to evaluate to true, it would try a different expression. After a few tries, if the FPG still could not find a suitable expression it would abandon the creation of the if/else block entirely.

Because it found an acceptable expression for the if’s condition, the FPG begins creating statements for the body of the if. At this point, however, the FPG must update its list of active states. Only a subset of its list of states will cause the if’s condition to evaluate to true, and those states are the only ones that will be affected by the statements within the if’s body. In this example, State A is affected by statements in the if’s body, and State B is not. State A is termed an *active* state within the block, and State B is termed an *inactive* state. Inactive states are shown grayed out in Figure 5-10. Within the if’s body, the FPG decides to create one assignment statement on line 17 and updates all active states to reflect its execution.

After the FPG completes the if's body, it decides to create an else block. Note that every if block may have an else if block and/or an else block, but neither are required. Before the FPG can create the else block, it must ensure that at least one of its test cases will cause the block to execute. It returns to the if's condition and searches through the active states to find one that causes the condition to evaluate to false. If it was unable to find one, it would abandon the else block (leaving the already-created if block intact). The FPG, however, finds that State B causes the if's condition to evaluate to false and proceeds with the creation of the else block. Within the block, it creates the pointer assignment statement on line 22 and updates all active states to reflect its execution.

After the if/else block, the FPG decides to create a dereferenced pointer assignment on line 27. Both states are active at this point, so the FPG updates both of them. Note, however, that the pointer "p" points to "x" in State A and to "y" in State B, so the statement causes the states to be updated differently. Finally, the FPG creates a final call to printf to print the values stored within each variable. The test case that began State A will cause "-5 9 -5" to be printed, and the test case that began State B will cause "-8 28 28" to be printed. These are the outputs the FPG will expect from the student's response to this problem (though this is not a realistic example because the program contains no functions other than main). Though the FPG knows what output to expect from each test case, it does not use this information to test students' solution programs. Instead, it compiles and runs for each test case the initial inlined program given to the student to obtain the "correct" output. This ensures that the outputs from the "correct" program and the student's solution are obtained from programs compiled on the same compiler and run on the same system.

Though it is not shown in the example of Figure 5-10, the FPG sometimes executes statements that have already been written. In loops, the FPG fills the body with statements and executes them, representing the first iteration of the loop. When all statements for the loop have been created, the FPG returns to the start of the loop to run the remaining iterations (of which there may be different numbers for different states), updating active states to reflect the statements' execution but creating no new statements. Similarly, when the FPG creates a call to a function, it creates and executes statements for the body of that function. When the FPG creates successive calls to that function, it executes the function's body but again creates no new statements.

6. Experiments and Experimental Data

To verify that the FPG is a viable tool that can be deployed in CS classrooms, two experiments were conducted. First, a random sampling of FPG problems was examined in order to determine the extent to which the FPG inserts meaningless or easily simplified code into its programs. Second, a student from a CS1 course was given the opportunity to solve FPG problems, and through a survey he was able to give feedback about the tool. The findings from these two experiments are presented in this chapter.

6.1. *Manual Examination of FPG Problems*

Before any FPG problems were presented to a student, an experiment was conducted to determine the extent to which the FPG generates meaningless or easily simplified code. The frequent appearance of such code has two effects on the quality of FPG problems:

- Code can be simplified or removed to reduce the token count, circumventing the FPG's check on the degree of function factoring
- Programs appear less realistic, reducing their impact on students

It is desirable that very little meaningless or easily simplified code appears in programs generated by the FPG, if any appears at all.

In this experiment, the FPG was used to generate thirty programs, three at each of the ten difficulty levels at which default FPG Parameter objects are available. Each program was examined by hand to locate any undesirable code, and the instances of such code were counted. An instance sometimes consisted of a single statement but often consisted of a few consecutive statements.

In the thirty programs that were examined, a total of fifty-eight instances of undesirable code were found. Of these, thirty-eight were instances of extra variables. An *extra variable* here refers to a local variable that was declared for a function but that had not been used within the code at the time the FPG decided to conclude the function's body. When the FPG decides to wrap up a function but has unused local variables, it creates assignment statements assigning to those variables at the end of the function body. An example is shown in Figure 6-1.

```
( 1) int funcA(int a) {  
( 2)     int b, c, d;  
( 3)  
( 4)     c = 9 - a;  
( 5)     for (b = 0; b < 9; b = b + 1) {  
( 6)         printf("%d", a);  
( 7)     }  
( 8)     d = c * a % 10 + b;  
( 9)     return d + b;  
(10) }
```

Figure 6-1: Function Containing an Extra Variable

In this example, the FPG creates the assignment statement on line 4 and the for loop on lines 5-7 before deciding to conclude the function. At that point, the local variables “b” and “c” have been used, but “d” has not. The variable “d” is considered an extra variable. As a result, the FPG creates the assignment statement on line 8 to initialize “d” and then uses it in the return statement on line 9. This avoids meaningless code in that the declaration for “d” cannot be deleted as the code currently appears. However, the expression that is assigned to “d” on line 8 could simply be substituted for “d” in the returned expression. The assignment on line 8 could then be deleted along with the declaration of “d”. Everywhere that this function is inlined in the initial program given

to the student, the student could perform this substitution and reduce the token count, possibly resulting in fewer functions needing to be factored to satisfy the grading mechanism. The frequency of the appearance of extra variables could be reduced by increasing the length of functions, giving the FPG more opportunity to create statements that utilize all the variables declared in a function.

Another seventeen instances of undesirable code found were similar instances of easily simplified code. These were not caused by a necessity to use a declared variable as is the case with extra variables, but rather these are simply the product of the FPG's random code generation. An example of such an instance is shown in Figure 6-2.

```
(1) int a, b;  
(2)  
(3) scanf("%d", &a);  
(4) for (b = 3; b > 0; b = b - 1) {  
(5)     a = a - 6;  
(6)     a = a - b;  
(7) }  
(8)  
(9) printf("%d %d", a, b);
```

Figure 6-2: Example of Easily Simplified Code

Consider the code in the body of the loop on lines 4-7. As was the case with extra variables, this code is not meaningless: the final value of “a” is changed by both statements in the loop, and the final value depends on the number of times the loop body is executed. These statements cannot simply be deleted from the program for these reasons. Again, however, the statements can be easily simplified by combining them into a single assignment statement, reducing the token count of the program.

Finally, the remaining three instances of undesirable code were combinable loops similar to the one shown in Figure 6-3.

```
(1) int a, b;  
(2)  
(3) scanf("%d", &b);  
(4) for (a = 0; a < 7; a = a + 1) {  
(5)     printf("%d", a + 4);  
(6) }  
(7) for (a = 0; a < 7; a = a + 1) {  
(8)     b = b + a;  
(9) }
```

Figure 6-3: Example of Combinable Loops

In this block, the two for loops on lines 4-9 each run the same constant number of times. Because the statements in the loop bodies are independent of each other, the assignment statement on line 8 could be inserted after the printf call on line 5. The loop on lines 7-9 could then be deleted from the program. It should be noted, however, that two out of the three pairs of combinable loops that were found ran different constant numbers of iterations, so combining them would be more difficult than this example illustrates.

Through this experiment, it was determined that appearances of meaningless code were extremely rare but that appearances of easily simplified code are relatively common. However, none of the thirty programs examined could have its token count reduced through code simplification to the point where function factoring was not needed to solve the problem. The FPG's problems were therefore deemed fit to be given to students to solve.

6.2. Student Use of the FPG Module

To test the FPG system with actual students, students nearing the completion of the first quarter of the introductory series of computer science courses at Cal Poly were allowed to attempt to solve FPG problems for extra credit in their class. One student elected to participate in this first trial of the FPG. After the completion of two FPG problems, the student was asked to take an online survey to provide feedback for the system. The text of the survey appears in the appendix of this report.

The student was asked in the third question of the survey to rate on a scale of one to five the different causes of the FPG grading his responses as “incorrect” according to how frequently he experienced them, where a rating of one corresponded to “never” and rating of five corresponded to “frequently”. The student rated “failed to compile” as the most frequent cause, assigning it a rating of five. “Failed test cases” followed with a rating of four, and “too many tokens” was given a rating of two. The frequency of compilation failures demonstrates the importance of intelligent compiler feedback, a feature provided by the IHS. The student’s rating of “failed test cases” and “too many tokens” follows from the fact that the FPG checks for proper input/output functionality before counting tokens, so it is to be expected that the FPG will report failed test cases more often than it will report too high of a token count.

In the fourth and fifth questions of the survey, the student was asked to rate the difficulty of understanding how he should go about solving FPG problems as well as the difficulty of actually solving the problems. On a scale of one to five where five was the most

difficult, the student assigned both questions a rating of four. The first of these is the most concerning; it is preferable that students do not find great difficulty in understanding what FPG problems ask of them. However, a demonstration from the instructor would go a long way in explaining how FPG problems are solved; the student did not receive such a demonstration. Because one of the stated goals of the IHS and the FPG is to lessen the hurdles students must overcome to understand CS concepts, the student's answer to the fifth question is also a bit concerning. The student, however, stated in the comments section that not knowing the extent to which the FPG expected programs to be factored was the most difficult aspect of the exercise. This suggests that the difficulty could be lessened by providing this information to the student along with the unfactored program, perhaps by listing the expected token count or the expected number of factored functions. Additionally, it should be noted that *some* difficulty is expected and perhaps even necessary for students to learn this concept.

The student was asked in the sixth question how helpful the FPG exercise would have been when he was first learning about functions. On a scale of one to five where five was the most helpful, the student assigned the exercise a rating of four. This suggests that the FPG has promise in teaching the concept of function factoring to students. The student commented in his answer to the following question that he could “see where [the FPG exercise] would help beginners starting out”. Additionally, he further commented that FPG problems were something he “could practice between in class lectures/labs [sic]” suggesting that the individual problems are small enough in scope that a student could work on them even in small windows of time.

Finally, the student made two suggestions for the FPG in his response to the last question: that the expected number of factored functions should be listed and that the initial unfactored code should be displayed in a second, non-modifiable window such that the student cannot lose track of it. Both of these suggestions could be implemented with minor tweaks to the FPG's interface to students through the IHS, and no changes to the overall design of the system would be needed.

The student's success with the FPG system and his responses to the survey show promise for the FPG. No major issues arose in this initial trial of the system, and the student stated that completing FPG problems could be very beneficial for beginning CS students. Though a single student is not a representative sample of all CS students, the results of this experiment demonstrate that the FPG is ready to be used in an introductory CS class to further exercise the system and to obtain additional data.

7. Summary and Conclusions

Over the last few decades, computer scientists have been attempting to automate parts of the process through which the next generation of computer scientists is educated. Though many tools for use in the CS classroom have been proposed, few of them have seen anything other than local use. More recently, computer science educators have attempted to tie greater functionality into intelligent tutoring systems that can be used throughout entire CS courses. This emerging area of research has shown promise but no such tutoring system has been widely adopted.

This report has documented a new tool, the Factoring Problem Generator (FPG). The FPG generates programming problems in which students must locate and extract blocks of repeated code into individual functions. This process of dividing the functionality of programs into distinct subroutines has been identified as one with which beginning CS students have considerable trouble. The FPG thus presents a new way for instructors to teach a difficult issue to students. The high configurability of the FPG allows it to be customized for any classroom, and the automated creation and grading of problems allows instructors to easily assign several small problems of increasing difficulty to students so that they may be slowly but surely guided through the learning process.

An initial examination of programs produced by the FPG demonstrated that the system produces programs containing occasional instances of undesirable (though not erroneous) code. These instances, however, were neither frequent enough nor severe enough to cause problems with the way students complete the exercises. One beginning CS student

completed a few FPG problems, and through an online survey it was found that he believed that working through the problems when he was first learning about functions would have been beneficial.

The FPG, then, is ready for use in CS classrooms. Though improvements can be made to the system (and some examples of these will be discussed later in this chapter), it is currently a viable component of the IHS framework. As contributors add functionality to that framework, it will hopefully offer enough variety and flexibility to take the IHS – and the FPG – to classrooms outside those at Cal Poly.

7.1. Contributions

This work has presented a new tool to assist instructors in teaching a difficult CS concept. The FPG enables instructors to automatically assign programming problems without having to create and grade each one by hand. This ease of use allows several small programs of increasing difficulty to be assigned, allowing students to gradually become more proficient at using functions and to avoid the shock of having to complete large programming assignments. The FPG offers ten preconfigured difficulty levels, but instructors can further customize the FPG's problems as needed. The length and number of functions, the number and types of parameters, the frequency of certain operators and kinds of statements, and the allowed data types can all be configured to meet individual needs.

This work has also introduced a new algorithm for generating code programmatically. The use of lists to keep track of variables that need to be initialized and variables that need to be read has resulted in a system that produces random code without errors and without meaningless statements. This code can contain assignment statements, if/else blocks, for and while loops, function calls, and statements utilizing arrays and pointers. The presented algorithm ensures that code containing any or all of these elements will consistently make use of declared variables, initialize variables before they are read, and read stored values before they are overwritten.

Finally, this work proposes the idea of generating test cases before generating code. The FPG uses random test cases to create several initial states for a program and then writes the program while simultaneously maintaining the states to reflect the program's execution. When an if/else block or loop is written, the FPG writes the branching conditions such that at least one state causes the branch to execute. This allows the FPG to generate programs with sets of test cases that result in complete code coverage when the programs are tested. Where programs cannot be tailored to satisfy random test cases, this method is not applicable. However, the field of automation in computer science will likely have other areas where random code will be generated, and this method can be applied in them.

7.2. Future Directions

As was seen through experimentation, the FPG produces programs in which appear occasional instances of undesirable code. The vast majority of these are instances where

code can be easily simplified, providing a possible avenue for students to reduce the program's overall token count without factoring functions as intended, thereby defeating the FPG's grading mechanism. A solution to this problem would be to apply basic block simplification to programs produced by the FPG. Such simplification is a well-researched problem as it is frequently used as an optimization when compiling programs. The simplification, then, could likely be applied to the FPG without a great deal of additional research or development.

Another case of undesirable code that was seen to be produced by the FPG, albeit more infrequently, is the case where loops running constant numbers of iterations could be combined into a single loop. An example of such code was shown in Figure 6-3. One solution to this issue is to create loops that run varying numbers of iterations, possibly by initializing the loop count variable with or checking against a parameter. Loops that are allowed to run differing numbers of iterations cannot be combined, so this problem would be avoided.

A final possible improvement to the FPG concerns its overall approach to teaching function factorization. Students are driven to extract functions from FPG programs by the requirement of meeting a reduced token count; in other words, functions are a useful tool in the FPG because they reduce code size. In embedded applications, for example, this motivation is a powerful one. But in other applications, reduced code size is often not the reason to extract code into functions. Rather, functions are frequently used to organize programs based on functionality and to make testing easier, or for no other

reason than to make programs easier to understand by human readers. As it stands, the FPG does not teach these motivations.

To remedy this, a larger modification of the FPG system would be needed. Because the FPG generates random code, it cannot teach students to organize programs based on their meaning. If, however, the FPG were to produce code that served some purpose, students could be asked to organize code into functions for reasons other than to reduce code size. For example, the FPG could be expanded to produce programs including greater I/O capabilities and support for strings. Programs could then contain blocks of related statements – blocks that process strings, blocks that perform mathematical calculations, blocks that process input from the user, and the like – allowing students to organize functions based on those relations.

Each of these suggestions has potential to further increase the usability and applicability of the FPG in CS classrooms. For now, the FPG demonstrates that the automation of many parts of a CS instructor's workload is indeed possible. As the field of automation in computer science education continues to grow and evolve, it will be interesting to see how its ideas will be applied in computer science classrooms.

Bibliography

- Aiken, A. *MOSS – A System for Detecting Software Plagiarism*. <http://theory.stanford.edu/~aiken/moss/> (accessed November 13, 2009).
- Akingbade, A. et al. "JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses." *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (ACM Press), 2003: 162-166.
- Allen, E. et al. "DrJava: A Lightweight Pedagogic Environment for Java." *SIGCSE Bulletin*, 2002: 137-141.
- Anderson, J. R. and E. Skwarecki. "The Automated Tutoring of Introductory Computer Programming." *Communications of the ACM*, 1986: 842-849.
- Baker, R. S. "Modeling and Understanding Students' Off-Task Behavior in Intelligent Tutoring Systems." *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007: 1059-1068.
- Becker, B. W. "Teaching CS1 with Karel the Robot in Java." *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education* (ACM Press), 2001: 50-54.
- Chen, X. et al. "Shared Information and Program Plagiarism Detection." *IEEE Transactions on Information Theory*, 2004: 1545-1551.
- Cohen, D. M. et al. "The AETG System: An Approach to Testing Based on Combinatorial Design." *IEEE Transactions on Software Engineering*, 1997: 437-444.
- Colin, S. et al. "Preamble Computation in Automated Test Case Generation Using Constraint Logic Programming: Research Articles." *Software Testing, Verification, and Reliability*, 2004: 213-235.
- Cooper, S. et al. "Using Animated 3D Graphics to Prepare Novices for CS1." *Computer Science Education*, 2003: 3-30.
- Dadgostar, F. et al. "Making Sense of Student Use of Nonverbal Cues for Intelligent Tutoring Systems." *Proceedings of the 17th Australia Conference on Computer-Human Interaction*, 2005: 1-4.
- Douce, C. et al. "Automatic Test-Based Assessment of Programming: A Review." *ACM Journal of Educational Resources in Computing*, 2005: 4.
- Edwards, S. H. "Improving Student Performance by Evaluating How Well Students Test Their Own Programs." *ACM Journal of Educational Resources in Computing*, 2003: 1.

- English, J. "Automated Assessment of GUI Programs Using JEWL." *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education* (ACM Press), 2004: 137-141.
- Goldman, K. J. "A Concepts-First Introduction to Computer Science." *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 2004: 432-436.
- Higgins, C. A. et al. "Automated Assessment and Experiences of Teaching Programming." *ACM Journal on Educational Resources in Computing*, 2005: 5.
- Hundhausen, C. and S. A. Douglas. "SALSA and ALVIS: A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations." *Proceedings of the IEEE International Symposium on Visual Languages* (IEEE Press), 2000: 67-68.
- Hundhausen, C. et al. "A Meta-Study of Algorithm Visualization Effectiveness." *Journal of Visual Languages and Computing*, 2002: 259-290.
- Jain, J. et al. "Experimental Evaluation of Animated-Verifying Object Viewers for Java." *Proceedings of the 2006 ACM Symposium on Software Visualization* (ACM Press), 2006: 27-36.
- Joy, M. et al. "The BOSS Online Submission and Assessment System." *ACM Journal of Educational Resources in Computing*, 2005: 2.
- Karavirta, V. et al. "MatrixPro – A Tool for On-The-Fly Demonstration of Data Structures and Algorithms." *Proceedings of the 3rd Program Visualization Workshop*, 2004: 26-33.
- Kölling, M. et al. "The BlueJ System and its Pedagogy." *Computer Science Education*, 2003: 249-268.
- Krebs, M. et al. "Student-Built Algorithm Visualizations for Assessment: Flexible Generation, Feedback and Grading." *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (ACM Press), 2005: 281-285.
- Kumar, A. N. "Generation of Problems, Answers, Grade, and Feedback – Case Study of a Fully Automated Tutor." *Journal on Educational Resources in Computing*, 2005: 3.
- Lahtinen, E. et al. "A Study of the Difficulties of Novice Programmers." *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ACM Press), 2005: 14-18.
- Malmi, L. et al. "Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2." *Informatics in Education*, 2004: 267-288.
- Malpohl, G. *JPlag – Detecting Software Plagiarism*. www.ipd.uni-karlsruhe.de/jplag (accessed November 13, 2009).

- Moreno, A. et al. "Visualizing Programs with Jeliot 3." *Proceedings of the Working Conference on Advanced Visual Interfaces* (ACM Press), 2004: 373-376.
- Naps, T. L. et al. "Exploring the Role of Visualization and Engagement in Computer Science Education." *Working Group Reports from the 2002 Conference on Innovation and Technology in Computer Science Education* (ACM Press), 2002: 131-152.
- Naps, T. L. "JHAVE – Supporting Algorithm Visualization." *IEEE Computer Graphics and Applications*, 2005: 49-55.
- Pattis, R. E. *Karel the Robot: A Gentle Introduction to the Art of Programming*. New York: John Wiley & Sons, Inc., 1981.
- Pears, A. et al. "A Survey of Literature on the Teaching of Introductory Programming." *Working Group Reports on Innovation and Technology in Computer Science Education* (ACM Press), 2007: 204-223.
- Pillay, N. "Developing Intelligent Programming Tutors for Novice Programmers." *SIGCSE Bulletin*, 2003: 78-82.
- Ragonis, N. and M. Ben-Ari. "On Understanding the Statics and Dynamics of Object-Oriented Programs." *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005: 226-230.
- Soh, L. "Incorporating an Intelligent Tutoring System into CS1." *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 2006: 486-490.
- Staley, C. "The Stair-Step Model for Teaching STEM Subjects." Teaching Computer Science. California Polytechnic State University. San Luis Obispo, CA. January 2009.
- Storey, M. et al. "Improving the Usability of Eclipse for Novice Programmers." *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange* (ACM Press), 2003: 35-39.
- Valentine, D. W. "CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings." *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 2004: 255-259.
- Weber, G. and P. Brusilovsky. "ELM-ART: An Adaptive Versatile System for Web-Based Instruction." *International Journal of Artificial Intelligence in Education*, 2001: 351-384.
- Wise, M. J. "YAP3: Improved Detection of Similarities in Computer Program and Other Texts." *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (ACM Press), 1996: 130-134.
- Zeller, A. and D. Lütkehaus. "DDD – A Free Graphical Front-End for UNIX Debuggers." *Proceedings of the Special Interest Group on Programming Languages*, 1996: 22-27.

Appendix – FPG Feedback Survey

This survey asks you eight questions about your experience with the function factoring exercise in Dr. Staley's online homework system. Your honest feedback is greatly appreciated!

1. How many function factoring problems did you successfully complete (receive a final score of 100%)?

2. How many times did you fail (receive a final score of 0%) in an attempt to solve a problem?

3. When your solutions to problems were returned by the autograder as incorrect, how often was it for each of the following reasons? If you never had a problem returned to you as incorrect, please skip this question.

	Never		Sometimes		Frequently
Failed test cases	___	___	___	___	___
Too many tokens	___	___	___	___	___
Failed to compile	___	___	___	___	___

4. When you first began solving these problems, how easy was it for you to figure out what you were supposed to do?

	Very Easy		Moderate		Very Difficult
Please choose:	___	___	___	___	___

Comments:

5. Once you understood what you were supposed to do, how difficult did you find the problems to solve?

	Very Easy		Moderate		Very Difficult
Please choose:	—	—	—	—	—

Comments:

6. How helpful would this exercise have been when you were first getting the idea about functions?

	Not At All Helpful		Somewhat Helpful		Extremely Helpful
Please choose:	—	—	—	—	—

Comments:

7. What did you like most about this exercise?

8. What about this exercise would you improve?