

Characterizing novice compilation behavior

Matthew C. Jadud

October 11th, 2003

Contents

1	Introduction	3
1.1	Motivation	3
1.1.1	Folk wisdom	3
1.1.2	History	4
1.1.3	Literature	4
1.2	The Project	5
1.3	Structure of the prospectus	6
2	Review of literature	8
2.1	Historic Perspectives	8
2.1.1	CORC	8
2.1.2	DITRAN	10
2.1.3	TOPPS & TOPPS II	10
2.1.4	NT & ST	11
2.1.5	PL/I	12
2.1.6	Relevance to our work	12
2.2	Pedagogic languages	13
2.2.1	LOGO	13
2.2.2	Pascal	14
2.2.3	Relevance to our work	15

2.3	Pedagogic programming environments	15
2.3.1	CAP [Pascal]	15
2.3.2	BlueJ [Java]	16
2.3.3	DrScheme [Scheme]	16
2.3.4	The CS1 Sandbox [C]	17
2.3.5	Relevance to our work	18
3	Experimental Design	19
3.1	Baseline behavior	19
3.2	Interventions	20
3.2.1	Single-case interventions	20
3.2.2	Withdrawal (<i>ABAB</i>) interventions	21
3.2.3	Multiple baseline interventions	21
3.3	Questions and Considerations	22
3.4	On-line protocols	23
4	Timeline	27
4.1	BlueJ	27
4.2	Alternate Environments	27
4.3	BlueJ*	29
4.4	Conferences	29
4.5	Coding, Analysis, and Writing	30
5	Structure of the Dissertation	31

Section 1

Introduction

We are interested in **novice compilation behavior**. By **novice**, we mean a student with little or no previous experience programming a computer. By **compilation**, we mean the programmer's process of tool use for converting their source code into something the computer can execute. By **behavior**, we mean the *what*, the *when*, and the *how* of compilation.

We choose to focus on the *behavior* of programming because it is directly observable: we can record with great precision when a student decides to test their most recent changes to a program. We cannot so easily record *why* a student made the changes they did, or why the student believes the most recent additions to their program will fix the errors they observed the last time they compiled their program.

1.1 Motivation

We believe there is a great deal to be learned in the study of novice compilation behaviors. Our motivation comes from folk wisdoms taken as truth in programming instruction, technological innovation, and inspiration from the literature.

1.1.1 Folk wisdom

We have observed that different students in introductory programming courses engage in different compilation behaviors; some students can be observed compiling their programs many, many more times than others. We assumed

that one kind of student was thinking about their problems, while another was (obviously!) making changes and recompiling while hoping for the best. This led us to make the comment (more than once) that *the students aren't even thinking about how to solve their problems... they're just letting the compiler think for them*.

We are not unique in this sentiment, as we have heard many of our colleagues say the same thing of their students. Yet, we have little empirical evidence to support the conclusion that either class of student is thinking more than the other.

1.1.2 History

The folk wisdom that fewer compilations means more thought is taking place is clearly rooted in the history of programming behavior. In the 1960's, writing and executing a program on a computer was not a trivial task. The program was often developed long-hand (in writing, with tools like a pencil and paper), and then punched onto cards. These cards would be submitted to the computer operator, who would feed them into the computer when time permitted, and eventually return the cards as well as the output of the program; this process could take one or more days to complete. The programmer would then find out about any typographical mistakes they made, as well as any other syntactic errors they may have had. Simply stated, programmers spent more time writing their programs during this era of computing than they do today.

For us, we see it as an obvious sign that compilation behavior has changed. We do not know if this is good or bad; we do know that it has not been researched extensively.

1.1.3 Literature

We are interested in one paper in particular: "Conditions of learning in novice programmers," by Perkins, Hancock, Hobbs, Marin, and Simmons at Harvard University. Published in 1986, their studies provided insight into the behavior and process of novice programmers.

Under normal instructional circumstances, some youngsters learn programming in BASIC or LOGO much better than others. Clinical investigations of novice programmers suggest that this happens in

part because different students bring different patterns of learning to the programming context. Many students disengage from the task whenever trouble occurs, neglect to track closely what their programs do by reading back the code as they write it, try to repair buggy programs by haphazardly tinkering with the code, or have difficulty breaking problems down into parts suitable for separate chunks of code.

Within this context of studying young programmers, Perkins, et. al.'s observation of *stoppers* and *movers* is a useful distinction. Stoppers are children who, when confronted with a problem, just give up. Movers, on the other hand, may get lost, but because they have forward motion, it is possible for them to solve their problems.

Stoppers and extreme movers can be viewed as being at endpoints of a continuum based on the ratio of time spent thinking (or time spent sitting in front of a terminal and not typing) to time spent entering and testing code. But this image of a continuum is in a way misleading. It suggests a distribution with most students in the middle while extreme stoppers and movers occupy the statistically rare tails. On the contrary, the descriptions of stoppers and movers are not caricatures of the norm. Extreme stoppers or movers are common.

Having witnessed this same behavior in our own university classrooms, we hope our work will provide empirical support for these observations.

1.2 The Project

We propose to carry out an **applied behavioral study** of novice compilation behavior. The *act of compiling* a program is observable, and clearly shaped by environmental factors. We propose a **behavioral** study because this is the simplest framework in which we can couch an examination of compilation. The study is **applied** because we propose to study novice compilation behavior in a natural environment as opposed to controlled, laboratory conditions.

Our first year will be spent observing and analyzing programs written in terminal (laboratory) sessions by novice programmers at the University of Kent. During these sessions, students use BlueJ, a programming environment for the Java programming language designed with pedagogic principles in mind;

environments of this nature are often referred to as *pedagogic programming environments*. Our goal in capturing when compilation took place, the code that was compiled, and the results of that compilation (errors, etc.) is to establish what constitutes normal, or *baseline* behavior for novice programmers using the BlueJ environment.

Based on our analysis of the data available (which will include the marks students receive for the programs they write, as well as their examination scores), we will consider the possibility of staging one or more *interventions* during the second year. Our conjecture is that compilation behavior may be a predictor for success—in Perkins’s terms, we can detect a *stopper* or a *mover* based on what and when the students compile—and will attempt to shape, or modify, the behavior of stoppers. Our hope is that we can encourage stoppers to keep moving by purely behavioral means, increasing their chances of completing the programming tasks set for them in their coursework.

We believe this work will be significant to the computer science education community for a three reasons. The first is purely methodological in nature: our work reintroduces behavioral research into the discipline. Many studies over the last twenty years have been cognitive in nature, focusing on the *why* of programming.[4, 26, 30] The more behavioral research of the sixties and seventies was lost, despite the fact that these small and simple studies provided a foundation for more complex research.[3, 7, 21] This is a problem, as cognitive studies tend to *assume* the behavioral.

Second, if successful, our tools will give us the ability to detect when a student has given up or is otherwise lost; this would be significant unto itself, as we could then instrument pedagogic programming environments like BlueJ to prompt students for help when they were stuck, or encourage them to slow down when they shifted too far into the “extreme mover” end of the spectrum.

Lastly, if we find no connections between strong students and their compilation behavior, then we have a significant result as well: compilation behavior is not a predictor for success in programming, a finding that would provide an empirical basis for supporting (or challenging) many methodological approaches to teaching programming at the introductory level.

1.3 Structure of the prospectus

Section 2 presents the literature we feel is particularly related to our work. We focused on literature regarding compilation tools and programming environments, as much of the development and analysis in these areas over the past

forty years has been driven by programmer behavior. For this reason, this literature tends to be most directly related to our work. Literature regarding the development of programming languages also plays a role in many cases, as languages represent the fundamental tool we use for writing programs and expressing ideas programmatically.

In section 3 we lay out the structure of an applied behavioral study, and discuss the nature of our particular exploration of novice programmers at the University of Kent.

Lastly, we present a timeline for the research in section 4, and the structure of the dissertation in section 5, which we expect to be complete during the summer of 2005.

Section 2

Review of literature

This section details a chronological review of the literature pertinent to our exploration of novice compilation behavior. Our focus on quantifiable programming behaviors align us closely with early work in languages and environments, when detailed numerical analysis of programmer behavior was more the norm. Qualitative efforts regarding the cognitive aspects of novice programming being carried out today provide us with umbrella theories under which we can explore foundations that have been otherwise ignored for the last twenty years.

2.1 Historic Perspectives

We are interested in how students go about the programming process, and are especially interested in their compilation behaviors while programming. Just as the automobile has (hindsight being 20/20) obviously changed the way people live and work every day, we can see that the thousand-fold increase in computational power since 1960 has changed the way people program computers. This obvious change in technology only further highlights our fundamental lack of understanding regarding the practice and teaching of programming.

2.1.1 CORC

The Cornell Computing Language project resulted in CORC, a language developed at Cornell University in the early sixties.[3] CORC is interesting both as a pedagogical language and for the debugging and compilation patterns exhibited by it's users—largely students and faculty in engineering, science,

and mathematics making use of this new and novel tool (the programmable computer) as part of their work:

CORC has been operating at the Cornell Computing Center since September, 1962 [1 year]. A procedure is used in which students deposit their programs at the Center, written on the special CORC forms. The personnel of the Computing Center key-punch the programs (without verification) and make the initial computer run. Card decks, original programs and computer output are then returned to the students in a large work room where card files, work tables, desk calculators, and key punches are provided. The students are responsible for making corrections in the card decks as necessary and placing these in a re-run drawer. The work is handled on a first-come-first-served basis and no attempt is made to segregate the problems by course. Except for a few times when the key-punching load peaked drastically a 24 hour turn-around time has been provided. Often a program submitted before 9 AM would be ready in late afternoon.

The Cornell Computing Center had at that time a Control Data Corporation 1604 and Burroughs B-220 general purpose computer. The CDC 1604, released in 1958, was the first computer designed under the direction of Seymour Cray. With its great size came great performance: the 1604 had a clock speed of 5 microseconds, making it one of the fastest machines of its generation.[1]

As of this writing [June 1963] almost 4000 programs have been submitted by more than 300 students in 20 different courses. The average number of passes to achieve acceptable operation is slightly more than two (somewhat more than 4000 reruns have been submitted). Equally significant, about half of the programs ran acceptably on the first pass. In interpreting these results one should bear in mind that for the majority of these students, instruction in CORC was an incidental inclusion in a mathematics or engineering course and that the instructor was simultaneously experiencing his first contact with automatic computation.

Definitions are at best contextual; for example, terms like “acceptable operation” and “correct,” as used above. Without explicit definitions, we cannot know exactly what the authors meant. We believe the authors generally meant “the program did what the programmer intended.”

While we have no additional usage statistics from the Cornell Computing Center, we know that 50% of the programs submitted by students did what the programmer intended when executed for the first time. On average, only one additional compilation was necessary to fix any remaining errors in a program.

2.1.2 DITRAN

DITRAN stands for Diagnostic FORTRAN; developed at the University of Madison, Wisconsin, it was first put into use in 1965, and reported in the Communications of the ACM in 1967 by Moulton and Muller. Like CORC, DITRAN was developed with pedagogical goals in mind, although it was felt that working with an existing language (rather than inventing a new one) was the appropriate approach. Programs submitted by all users to the University of Wisconsin Computing Center (UWCC) averaged 38 statements in length. Like the Cornell Computing Center, Wisconsin made use of the CDC 1604.[21]

To evaluate the usage of the UWCC, programs were stored for later study. Moulton and Muller provide some summary analyses of the programs collected during March, April, and May of 1966. Of the 5158 programs compiled during this time, 1859 of them had compilation errors, with an average 3.8 errors per program. 64% of the programs submitted at the University of Wisconsin were syntactically correct on their first submission.[21]

2.1.3 TOPPS & TOPPS II

J.D. Gannon and J. J. Horning were concerned with software reliability in the mid-1970's. While their work was, and continues to be, concerned with the production of reliable software on a systems level, they carried out some very interesting experiments in language design at the University of Toronto in 1975.

Programming languages should lead to confidence in the correctness of programs. They should aid the programmer not only in the coding process, but also in the process of (formally or informally) proving programs correct and maintaining programs.

Ultimately, our most powerful weapon against incorrect programs lies in the understanding of those who write and check them.

Gannon carried out an experiment regarding the use of TOPPS, a small pedagogic language (developed at the University of Toronto) with constructs for

supporting concurrency.[11] He was interested in whether the evolution of TOPPS (into TOPPS II) would enhance the reliability of code produced by students.

For the purposes of this study, a language was judged to enhance the reliability of software if the [syntactic] errors committed by its users were less frequent and less persistent.

Gannon found that the number of syntactic errors in programs generated by novices, as well as the number of compilations required to fix those errors (error persistence) could be reduced by selectively evolving the design of TOPPS into TOPPS II.

2.1.4 NT & ST

Examining the languages NT and ST (non-typed and strongly-typed), a pair of languages developed at the University of Maryland for research and teaching, Gannon and Dunsmore evolved their research regarding NT and ST into a more general search for linguistic factors contributing to the complexity and reliability of software.[7] Their research is an early example of the analysis of a novice's program from one compilation event to the next:

The initial version of a program represents a major commitment on the part of the programmer. Barring a complete rewrite, the programmer creates intermediate versions by making alterations to the evolving form of the initial version. Difficult or inappropriate language feature choices made in the initial version can lead to a plethora of program changes. On the other hand, a well-constructed initial version may either be initially correct or require only minimal changes.

They found that program complexity in the two languages appeared to be influenced by several factors, all of which were tagged for further exploration. In comparing NT and ST, they found that the average nesting depth, the amount of work involved in handling data structures, variable references, the number of live variables at any one time, and other simple factors seemed to contribute significantly to the complexity of the programs written by students.

Research regarding TOPPS and the comparison of NT and ST is important in the history of pedagogic programming languages. The work carried out by

Gannon and his colleagues is an early exemplar of methodologically rigorous, pedagogically oriented, user-directed language evolution. The design criteria they employed were generally intended to reduce complexity, be it type checking, declarative redundancy (eg. `private int x`, clearly restricting the scope and type of the variable `x`), or the way segments of code are composed to form a complete program; what role his research has played in the design of modern languages, if any, is unknown.

2.1.5 PL/I

Marvin Zelkowitz and his colleagues instrumented the PL/I compiler running on the University of Maryland's Univac 1108 to capture information about every program compiled and executed on the system, including the complete source code of each program.[35] Work by researchers like Gannon, et. al. and Zelkowitz can be contrasted with earlier studies (like those done with CORC and DITRAN): typically 40% to 50% of all programs submitted in the Gannon and Zelkowitz studies included at least one compile-time error.

There is almost no shared context between these studies and earlier ones, however: at Cornell and Wisconsin, programmers were scientists and students just learning to use a new tool, programs were short (roughly 30-40 statements), and there was a high cost (in terms of time and money) to using the computer. A decade later, Zelkowitz reports program lengths on the order of 200–500 statements (written by *computer science* students), and error rates that are significantly higher. While we can compare these numbers, we must always remember that the contextual differences involved can be significant.

2.1.6 Relevance to our work

The field of programming languages and compilers has an interesting history, shaped both by developments in mathematics and hardware; the changing nature of the computer itself has shaped and affected what is and is not possible in the design and implementation of languages and their compilers.

Studies regarding the use of CORC and DITRAN provide a historical point of reference regarding the use of languages and tools with novices that is very unlike the programming environments novices encounter today. Additionally, the thread traced from these early language implementations through to more modern tools provides us with a sense of history, a context for our own studies and analysis of novice compilation behavior.

The work of Gannon and his colleagues explored methodological approaches that had not been used previously, most likely due to a lack of computational resources. First, Gannon explored “program change,” which is how programs evolved over successive edit and compile cycles—an idea of interest to our own explorations.[14] Their notion of *error persistence* is a compelling measure in the context of compilation behavior: given the existence of an error in a novice’s code, how many edit → compile → run cycles are necessary before the error is removed by the programmer? Gannon found that a drop in error persistence correlated with the evolution of NT to ST, meaning novices using ST required fewer compilations to find and correct errors in their programs.

2.2 Pedagogic languages

Since the time there were programming languages, there have been concerns about whether one language was more “learnable” than another, or what could be done to make a language more comprehensible to beginners. Thousands of languages later, questions regarding language choice and its impact on how a novice learns to program continue.

2.2.1 LOGO

LOGO was developed in 1967 by Daniel Bobrow, Wallace Feurzeig, and Seymour Papert, and became part of an ongoing focus regarding children, programming, and learning at the MIT Artificial Intelligence Laboratory. Today, LOGO is in wide distribution, but rarely sees usage beyond elementary or primary school contexts in favour of “real” programming languages.

The majority of the work surrounding LOGO usage is qualitative and broad-sweeping, involving the implementation of entire curricula centered on building, programming, and exploring with LOGO.[23, 24] In contrast, Sharon Carver carried out a series of comparative studies of novice debugging behavior in 1988, probing the role of constructivism in the context of novice debugging process and behaviors.

In collaboration with Seymour Papert’s research group at MIT, Carver also studied whether 5th grade students given exhaustive unstructured LOGO experience (roughly 200 hours) acquire effective debugging skills. Subjects ... were unable to gather effective clues about the identity and location of the bug; therefore, they relied heavily on serial search.[15]

By “serial search,” Carver is referring to a process by which students go line-by-line through a program, as opposed to using heuristics informed by the structure of the language or some sort of other problem-solving methodology to debug their programs. Her research into novice debugging strategies is of interest, as the debugging process is part of the edit→compile→run cycle, and the novice programming behaviors observed in her research may present themselves in our research as well.

2.2.2 Pascal

Pascal, developed in 1970, continues to play a role in computer science education today. Developed by Niklaus Wirth at ETH Zürich, it was intended as an evolutionary step forward from Algol. Pascal introduced the ability to define complex datatypes from simpler forms, as well as dynamic structures, eliminating the need for programmers to statically define array bounds.[25] Pascal also supported compilation to P-code, an intermediate, architecture-neutral language which could easily be ported to new platforms.

Because of Pascal’s rapid and widespread acceptance into the introductory programming context, the learning of programming in Pascal became the subject of a number of studies. James Bonar, James Sporher, and Elliot Soloway explored conceptual bugs in novice programs, and proposed a theory of program construction and comprehension based on *programming plans*: stereotypical, canned solutions which form basic building blocks for analyzing programs and constructing programs.[29, 32] These “plans” were put forward as a cognitive theory of how programmers organize knowledge and information about a problem before and during the programming process.

Their notion of “bugs” was limited to those which were conceptual in nature; they were interested in *semantic*, as opposed to syntactic, errors. However, even this is too narrow an interpretation; Soloway et. al. had a much broader definition of what constituted a *bug*, based on their theories of how novices and experts went about the task of writing programs. To find these *conceptual bugs*, an expert first had to identify the plans and goals expressed in an ideal solution to the problem being tackled.[10] Deviation from this plan is what Soloway et. al. considered a *bug*—something that is not directly observable, but can only be found after identifying the *plans* embedded in a program.

Using their theory of goals and plans, Sporher and Soloway characterized 183 programs written by 61 novices in Pascal. The significant result of this work was that of the programs studied, 20% of the kinds (types, categories) of bug encountered represented 55% of all observed bugs.[31] This work was later

followed up by Cunniff and Taylor in a visual representation of Pascal they called FPL; their results, given a relatively small subject pool, supported the results of the earlier work done by Sporher and Soloway.[4]

2.2.3 Relevance to our work

Studies following from Soloway, et. al. tend to focus on the conceptual errors in novice programs at the exclusion of all other kinds of errors (type errors, syntax errors, etc.). They ignore everything that takes place before the *first successful compilation* of a piece of code, the theory being that the first syntax-error free program contains the essence of a programmer's misunderstandings, without the mechanical errors in syntax that might have taken place previously.

Carver's work into novice debugging behaviors tells us something concrete about young novice programmers learning LOGO in an unstructured, exploratory setting: they compile (and therefore execute) their code a lot. Despite LOGO being designed as a language for use in the classroom, it appears LOGO does not provide any significant support (either due to the nature of the language, or the environments used to programming in LOGO) for the novice engaged in an edit → compile → run sequence.

2.3 Pedagogic programming environments

The term *pedagogic programming environment* means many things. It could be a tool for assisting students to write programs; it could be a tool students use to analyze programs they've written; perhaps it is an expert system, or "wizard," that guides students through the programming process. More radically, it might mean a classroom environment designed for pair programming, or exploring robotics in a unique or particularly natural way. Every one of these meanings has been used in the literature, in one form or another.

2.3.1 CAP [Pascal]

CAP, written by Tom Schorsch, was an automated self-assessment tool that students at the United States Air Force Academy were required to run against their programs before submitting them. The goal was to provide syntactic and (limited) semantic checking for the programmer in language they would

understand.[27] While students appreciated the extra help (as reported in exit surveys), there were those who reported that they eventually got lazy in their programming: they knew CAP would find their errors for them.

Unfortunately, we believe that many students began using CAP as a crutch to merely get by. Rather than incorporating the required programming style rules into their programming habits, some students ignore style altogether knowing that CAP will annotate their code with all the corrections that are necessary. These students have become dependent on CAP and could not program to our standards without it. This is the opposite effect that we wanted.[27]

2.3.2 BlueJ [Java]

BlueJ is a pedagogical programming environment for Java, and to some degree, the object of our study. Work towards what we now know as BlueJ began in the early 90's; Kölling, Koch, and Rosenberg started by laying out their requirements for a first year, object-oriented teaching language. After evaluating C++, Smalltalk, Eiffel, and Sather for use as a student's first language and introduction to object-oriented programming, the authors felt it was necessary to develop a *new* language, designed to support novices in the programming endeavor. This language was called Blue.[16]

To support novices programming in this new object-oriented language, Kölling et. al. developed an environment to support students learning about classes, objects, inheritance, providing a UML-like visual environment to explore these ideas. Unfortunately, the effectiveness of this new language in combination with it's supporting environment isn't known: in 2001 the *Blue* environment becomes *BlueJ*. For reasons that are not made clear in the literature, Blue (a language designed with clear pedagogic intent) is dropped, and Java is plugged into the environment instead. In dropping Blue, the focus of later papers shifted to the role of the programming environment only, as opposed to the interplay of the language and the environment taken in combination.[17, 20]

2.3.3 DrScheme [Scheme]

DrScheme, developed by the Programming Language Team (formerly) at Rice, (now) at Brown University, Northeastern University, and the University of Utah, provides novice programmers an environment for exploring programming in the Scheme programming language.[8] DrScheme is particularly inter-

esting from a linguistic point of view, as the Scheme programming language has been broken into a series of increasingly powerful subsets of the full language. Students start with “Beginning Scheme,” and progress through “Intermediate,” “Advanced,” and eventually “Full Scheme.” At the earlier levels, error messages are tailored for the novice, providing as much support as possible for the beginner.

2.3.4 The CS1 Sandbox [C]

The CS1 Sandbox, developed by Peter DePasquale, is a simplified editing environment for a subset of the C programming language. The Sandbox provides a minimal interface and comprehensive help system to support novice programmers in their first exposure to the C programming language.[5] The Thetis project, developed at Stanford in the early nineties, also supported for novices working in C, but provided an interpretation and interaction environment very similar to DrScheme.[9] What separates the Sandbox from these other projects is the significant amount of data collected and analyzed regarding its usage in the introductory programming classroom at the university level.

DePasquale compared students learning to program while using subsets of the C programming language (a la DrScheme) with students exposed to the full language (both in the Sandbox environment and Microsoft’s Visual Studio .NET). Of particular interest are his findings regarding the compilation behavior of novices using the Sandbox environment: both in and out of the laboratory, novices who were programming with a limited subset of the language required fewer compilation and execution attempts to produce a working program than those using the full language. Simply stated, DePasquale provides empirical evidence that reducing the complexity of the language and environment (as well as tailoring the help to that reduced language) has an effect on the number of compilation events required to produce working code.[5]

Additionally, students were surveyed upon completion of the study, and were asked about their compilation behaviors while programming in whatever environment they were using. Students using all three environments reported similar figures; however, all the students are typically wrong about how much “work” (measured in terms of compilation and execution events) they have done:

Not only does the automatically collected data indicate a clearly decreasing mean number of compilations across the three projects, but the number of actual (automatically collected) values is between 1.5

and 2.5 times more than they reported. Clearly, the subjects seem to have a perception problem as it relates to the amount of work they are performing. That is, assuming the self-reported numbers are not intentionally deflated by the subjects (in an attempt to gain favor with this researcher), the subjects seem to be performing nearly twice as much “work” as they realize.

2.3.5 Relevance to our work

While recent trends seem to indicate that educators are seeing greater value in environments designed to support novice programmers, and many tools have been developed to this end, there is very little *research* into their effectiveness as tools for supporting the learning and practice of programming by novices.

CAP, while only a code-checking tool, tells us something about how novices behave when their compiler includes rich error reporting tools. Some take it as a challenge to reduce the number of errors they make, while others come to rely on the tool. Given that students in the sixties generally submitted correct programs on their first compilation/execution run (CORC, DITRAN), CAP provides us a very different picture of novice compiler usage.

The entire field of pedagogic environments is clearly applicable to our work, but the lack of research into the use of these tools by novices in the classroom setting is disappointing. Clearly, as demonstrated by explorations like those carried out by Schorsch and DePasquale, there is a great deal to be learned about novice programmers and the environments we construct to support them.

Section 3

Experimental Design

We propose to carry out an **applied behavioral study** of novice compilation behavior. The *act of compiling* a program is observable, and clearly shaped by environmental factors. We propose a **behavioral** study because this is the simplest framework in which we can couch an examination of compilation. The study is **applied** because we propose to study novice compilation behavior in a natural environment as opposed to controlled, laboratory conditions.

Our research will involve first observing the students' natural, or baseline behavior. Based on these observations, we may devise one or more interventions to explore and shape that behavior. What follows are definitions and examples of each of the primary components of an applied behavioral study, and as well as the particular difficulties we will face in determining baseline novice compilation behavior.

3.1 Baseline behavior

Normal. Typical. Everyday. These are all reasonable, one word definitions of baseline behavior. Our goal is to use well-defined, quantitative measures to describe the typical compilation behavior of novice programmers. This places our efforts (methodologically) between the statistical analyses employed in the sixties and seventies to evaluate the language and tools, and the more qualitative approaches employed by cognitive researchers of the eighties and nineties.

From the sixties and into the early seventies, we see a great deal of quantitative work, where numerical analyses of programmer behavior were the norm.[7, 11] In the eighties and nineties, qualitative work exploring cognitive aspects of novice programming behavior became the norm.[15, 31] While

we propose to begin our explorations with automated analyses of programs written by novices, an applied behavioral study has a fundamentally dualistic (qualitative and quantitative) methodological nature.

In approaching this study, we have one significant assumption about novice programmers and their interaction with the compiler: **success matters**. We believe it is desirable for the majority of a novice's compilation events to be successful. We assume that a student whose code compiles successfully the majority of the time is not only writing syntactically correct code, but also *logically* correct code. Our implication is that compilation behavior can be employed as an indicator of "learning" in the context of an introductory programming course.

3.2 Interventions

An applied behavioral study is often employed as part of a larger, behavior shaping process; once baseline behavior is identified for one or more individuals, an intervention is employed in an attempt to change the subject's current behavior. The methods employed in shaping the subject's behavior in any one intervention are based on those pioneered by B.F. Skinner in his research into operant conditioning.[28]

We cannot, at this time, speak to the nature of interventions regarding novice compilation behavior that we might carry out in a classroom setting with students using BlueJ. The type of intervention, or whether one is even warranted at all, will be decided after we have developed some notion of what constitutes baseline behavior in our target population. It is likely, however, that any intervention we do carry out will take the form of one or more of the most common experimental designs in behavioral studies: single-case, withdrawal, or multiple baseline intervention.

3.2.1 Single-case interventions

The goal of a single-case experimental design is to determine if the application of a treatment or intervention modifies the target behavior.[19] For example, we might observe one student who has a very high rate of compilation during terminal sessions (mostly unsuccessful), while generating very little code overall. We could encourage this student to ask more questions of their peers, or encourage the instructor to "drop by" this student's station more often to see how things are going. In either case, we would be looking for positive

correlation between the introduction of new stimuli, and their effect on the student's existing behavior.

3.2.2 Withdrawal (*ABAB*) interventions

An *ABAB* study involves multiple applications of a treatment interspersed with periods of no treatment, allowing relaxation back towards natural (baseline) behavior to take place. In an *ABAB* study, would hope to see baseline behavior change under treatment, and expect some *extinction* of the new behavior when the treatment is removed. Typically, after multiple treatments and periods of extinction, the effect of the treatment would become more effective, and the rate of extinction will decrease to the point that a new baseline behavior is adopted without the presence of the treatment.

3.2.3 Multiple baseline interventions

Multiple baseline studies involve changing more than one variable in an experiment, either the number of environment, stimuli, or number of individuals.

Environment People behave differently in different environments. One way to observe the effect of environment would be to observe one student's programming and compilation behavior both in and out of class. Another approach would be to collect baseline data at multiple educational institutions: our research instruments for gathering data are intended to be simple and portable to support this possibility.

Stimuli After performing single-case interventions with several students, we may choose to apply two or more interventions to a single student to see if we can achieve a cumulative effect in the modification of their behavior.

Individuals After identifying a particularly effective single-case or *ABAB* design, we could apply it broadly to an entire classroom; our goal in this case would be to see (at worst) no change in baseline, and at best an improvement in baseline behavior across multiple individuals.

It is likely that our experimental design will involve a combination of single-case, *ABAB*, and multiple baseline studies. We would begin with individual single-case or *ABAB* investigations, and perhaps make a generalization that at some point *all* students might benefit from a similar intervention. Or, more likely, there are some students who do, and some who don't, and our treatment

(assuming it is successful) will be non-invasive to students who don't exhibit the problematic behavior.

3.3 Questions and Considerations

A critical part of the environment, in addition to the tool used to develop code, is the language itself. There is hardly consensus amongst the computer science education community as to whether programming is even required in the first year of a computing degree, let alone which language would be most appropriate.[2, 6, 12, 13, 18, 22, 34] This is because computing, unlike mathematics, physics, chemistry, psychology, and other hard and soft sciences, lacks a single vocabulary for expressing concepts to beginners—every language introduces similar concepts in different ways, offering potentially radical views of the same idea.

With respect to compilation behavior, the features of the language effect how the student interacts with it while programming and compiling. Inspired by Gannon and Dunsmore, does the typing (strong vs. dynamic) effect compilation behavior?[7] Put simply, are type errors possible at compile time? Is the language lexically or dynamically scoped? What paradigm is the language most closely associated with (E.g. functional, procedural, object-oriented, declarative, visual, pseudocode, or pedagogically motivated)? Is the language interpreted?¹

In addition to the nature of the language, there are many additional factors related to the programming and learning environment that we must at least acknowledge in our observations. Do the students work alone or with a partner? Is their programming being done at home, or in a university computing laboratory? What kind of programming environment are they using—is it textual (Emacs, vi), pedagogic (BlueJ, DrScheme), small (PFE, JEdit), or an industrial strength environment (MS Visual Studio, Borland JBuilder)? Is most of the student's classroom instruction lecture- or laboratory-based? What is the typical size of the classroom environment? Are they working with state-of-the-art hardware, or is it cobbled together with bailing wire and sealing wax?

Any and all of these factors may play a role in novice programming behavior. Our initial work on defining baseline behavior will take place at the University of Kent; for the first year of research, we will only be observing students in

¹This may seem to be antithetical to the study of compilation behavior, but interpreted languages go through compile-time stages. For example, modern Scheme implementations like PLT and Chez Scheme have a complex series of macro- and module-expansion phases, during which any number of errors can be thrown before execution takes place.

one module, an introduction to object-oriented programming. The students in this course attend two lectures per week (roughly 200 students), and have one weekly terminal session, where 12-18 students work on programming tasks with an instructor present to assist and guide their efforts. In the laboratory sessions, students will be working in the BlueJ programming environment.

Our goal, within this constrained framework, is to develop a set of complete, unambiguous, replicable instruments for capturing data regarding student compilation behavior in the context of their interactions with their programming environment. The majority of our research will take place through the use of **on-line protocols**.

3.4 On-line protocols

On-line protocols are those that are carried out automatically by the computer. While we will collect data from many machines simultaneously, each client machine is identical: a Windows 2000 PC, standardized and controlled by the University, all running the same version of BlueJ . The clients are connected to a local area network, with limited access to the Internet as a whole; they report their data, collected from the BlueJ environment to our server via a local area network where it is stored for later analysis. We intend to focus on three kinds of information, each captured whenever a student invokes the compiler: timing data, their program at the time of compilation, and any compiler errors returned as a result of the invocation.

Timing

We want to know when a student invokes the compiler. This involves storing the time of compilation as reported by the client computer, as well as the time reported by the server when notice of the compilation takes place.² We capture both for a very simple reason: any error in the clock of a client will make it inconsistent with the clocks of all other clients. We can correct for this kind of inconsistency (within a reasonable margin of error) if we store the time each piece of data is received by the server, as well as the time the client reports sending their message.

Taken alone, timing data may be interesting. Assuming we know when the BlueJ session began, we can explore a large number of compilation behaviors for each student:

- How much time passes in each terminal session before the student's first

²In all cases, times will be stored as the number of seconds since January 1, 1970.

compile?

- How much time passes between compiles?
- How consistent (regular, periodic) is compilation?
- Does the student compile “a lot” compared to their peers?
- Does the student compile infrequently compared to their peers?
- Is compilation regular, perhaps denoting a consistent amount of programming followed by correction of errors?
- Does compilation come in infrequent bursts, implying a long period of development followed by the correction of multiple errors?
- Does the student’s compilation behavior seem to change over the course of a single session?
- Does the student’s compilation behavior, based solely on timing information, remain consistent over the course of a term? (Do compilation timings “fingerprint” a student?)
- Does the student’s compilation behavior, based solely on timing information, change significantly over the course of a term? What about from one programming environment to another?

Code deltas

In addition to information about *when* a student compiles, we want to know *what* a student compiles. In particular, we are interested in the changes a student makes between each compilation event. Inbetween any two compile events (call them C_n and C_{n+1}), the student will make zero or more changes to their code; we refer to the collection of these changes as a *code delta*, the value of which is δ .

$$\delta = C_{n+1} - C_n \quad (3.1)$$

We can quantify a code delta in many different ways, and each method may present new and interesting information. We can imagine very crude, even simplistic metrics for measuring how a student’s code has changed between compiles, as well as complex methods that take into account the structure and meaning of the Java programming language.

In the crudest of terms, we can make δ a discrete value, where

$$\delta = \begin{cases} 1 & \text{if } C_{n+1} \text{ is } > \text{ than } C_n \\ 0 & \text{if no changes were made} \\ -1 & \text{if } C_{n+1} \text{ is } < \text{ than } C_n \end{cases}$$

While this gives us no sense of the magnitude of change between compiles, it does tell us whether the student added or removed code in the process.

To add magnitude to the equation, we can take the Levenstein (STRING-EDIT) distance between C_n and C_{n+1} ; STRING-EDIT gives us the number of characters that must be added, changed, or removed to convert the program at C_n to the program at the time of compile C_{n+1} . [33] For example, the strings “why me” and “try me” have a STRING-EDIT distance of two: we must modify two characters to convert the former into the latter. Combined with our crudest calculation of δ , this gives us a sense of how much code the student added or removed between compiles.

The students in this study are editing programs, not just plain text; those programs have a well-defined grammar that we can exploit in our analysis. We can parse the code into a tree, and calculate how many nodes of the parse tree must be added, renamed, or deleted to convert one piece of code to another: essentially, a STRING-EDIT distance on trees (TREE-EDIT). If we ignore the labels on the nodes (we set the cost of renaming a node to zero), then we can measure how much structural change took place between compiles. This can give us a sense for whether the student is making structural changes (E.g. adding an `else` clause to an `if` statement), or working with the existing structure (E.g. renaming variables).

Using one or more of these values for δ , as well as our information regarding when compilation took place, a whole new class of questions about student compilation behavior can be developed:

- What is the average size of δ ?
- Does the student tend to grow their code consistently in size between compiles?
- Do students code-and-correct, or save lots of corrections for when they think they are “done?”
- Does volume of change correlate with time between compiles? For example, does a large amount of time between compiles imply a large amount of change?

- How much code do students write before compiling for the first time?
- As with our questions about timing, are there patterns in the volume of changes that take place over the course of a project?
- As with our questions about timing, can the volume of change over the course of a project “fingerprint” a student?
- Taken in combination, do timing and change provide enough information to identify a student? Classify them (E.g. “one of these things is not like the others...”).

Compilation: *success or failure*

If a student compiles their code without errors, this is a successful compile. If the student compiles and there are errors, it is an unsuccessful compile. We have defined these as the two possible consequences of a student’s compilation behavior. As part of our baseline analysis, we are interested in this binary outcome; because of the nature of the BlueJ programming environment, we know the student will only receive one error message at a time as a result of their compilation.

All of the questions we have asked about timing and the size of δ can be re-examined when we know the compilation outcome. We can examine compilation success and failure rates in broad classes of compilation behavior; for example, do students who compile very frequently with small number of changes to their code *succeed* more often than students who compile infrequently after making many significant changes? Does the compilation pattern or typical volume of change correlate with success or failure? These are purely exploratory questions: we have no guides in this area. Compilation behavior, and whether it results in successful or unsuccessful compilation events, is a very poorly understood aspect of novice programmer behavior.

Section 4

Timeline

Our timeline (Figure 4.1) is broken up into five tracks.

4.1 BlueJ

The primary focus of our study is novice programmers using the BlueJ programming environment. We intend to collect data regarding student usage during the 2003-2004 academic year as well as during 2004-2005.

4.2 Alternate Environments

As students grow more confident in their programming, it is not uncommon for them to use environments other than BlueJ in conjunction with compiling their code with `javac` from the command line. PFE is one editor commonly used (on Windows) by students at Kent; Emacs and `vi` are both popular as well.¹

While there are some difficulties in instrumenting other environments, and we can't claim that any variables in our study have been held particularly constant, comparing students who used BlueJ in our initial studies with students using other editors and compilation environments represents a great source of new information. It is even possible that we will be able to track the compilation behavior of students from the first term of study (using BlueJ) into the

¹Thread from local newsgroup, `ukc.cs.cs1`, timestamped Tue, 23 Sep 2003 13:31:48 +0100.

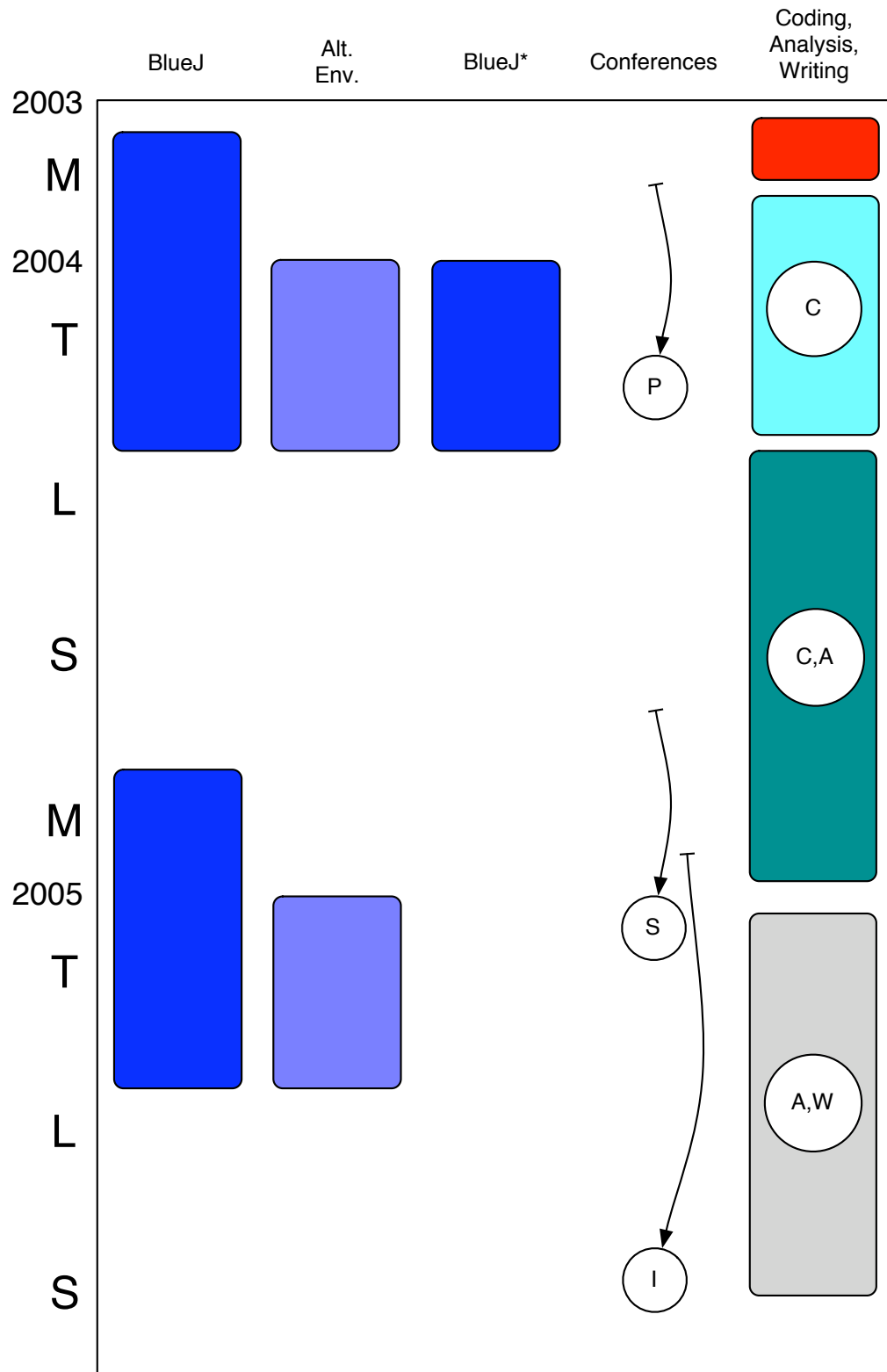


Figure 4.1: Novice compilation behavior research timeline.

second term (using some other environment), looking for baseline behavioral changes when the environment changes.

4.3 BlueJ*

We may have an opportunity in the second term to capture data from students at Cambridge, who will be taking their first course in Java; the term previous is taught in ML. What is particularly interesting is that they will, in some cases, be doing laboratory work that is identical to that of first-year at the University of Kent.

This external dataset gives us a new population using the same tools, in some cases writing the same programs. While keeping the many aspects of the environment “constant,” it varies the population, giving us a second baseline to compare to.²

4.4 Conferences

We currently see three conferences in the near future that will present obvious publication opportunities.

PPIG The Psychology of Programming Interest Group. In particular, our intent is to present the thesis and methodological approach using related, but externally obtained data.

SIGCSE The ACM Special Interest Group in Computer Science Education. The conference typically has a submission date at the end of August or early September; here, we will likely present initial findings from the first year, and future directions for analysis as we go into our second.

ITiCSE The European counterpart to SIGCSE, Innovation and Technology in Computer Science Education. Given the timing, this will be a summary of conclusions and findings.

Other conference opportunities present themselves both in the ACM and IEEE, most likely in the realm of software metrics. Given that we are employing

²In laboratory studies, the term *constant* might have a different meaning, where this is some nominal notion of control. In applied behavioral studies, we accept that each population may be unique, but we suspect that general, common behaviors will emerge.

similar strategies to this community (while working towards a different end), it is likely our work will be of interest to them.

4.5 Coding, Analysis, and Writing

The first two months of the autumn term in 2003 will be spent rapidly prototyping ideas on an externally obtained set of data similar to our own. The tools and techniques developed during this initial exploration will be 90% portable to data collected here at the University of Kent.

Our intention is to develop the majority of the tools to support analysis by the end of the 2003-2004 academic year. The summer of 2004 will be spent working with the data collected during the first year, writing tools targeted at specific questions our general analysis and exploration discover, and preparing for any changes that must be made going into the 2004-2005 academic year.

The dissertation will be completed during the second half of 2004-2005, to be submitted by the end of summer.

Section 5

Structure of the Dissertation

The dissertation will follow a traditional structure for research of this nature.

- i Acknowledgments
- ii List of tables
- iii List of figures
- 1 Introduction
- 2 Review of Literature
 - (a) Radical Behaviorism
 - (b) Compilation Behaviors
- 3 Experimental Design
- 4 Experimental Results
- 5 Discussion of Results
- 6 Conclusion
 - (a) Contributions
 - (b) Observations
 - (c) Future Work
- 7 Bibliography
- 8 Appendices

Bibliography

- [1] Cray research virtual museum, control data 1604, 2003.
- [2] Susan S. Brilliant and Timothy R. Wiseman. The first programming paradigm and language dilemma. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 338–342. ACM Press, 1996.
- [3] R. W. Conway and W. L. Maxwell. Corcthe cornell computing language. *Communications of the ACM*, 6(6):317–321, 1963.
- [4] N. Cunniff, R. P. Taylor, and J. B. Black. Does programming language affect the type of conceptual bugs in beginners' programs? a comparison of fpl and pascal. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182. ACM Press, 1986.
- [5] Peter J. DePasquale. *Implications on the Learning of Programming Through the Implementation of Subsets in Program Development Environments*. PhD thesis, Virginia Polytechnic Institute and State University, July 2003.
- [6] Roger Duke, Eric Salzman, Jay Burmeister, Josiah Poon, and Leesa Murray. Teaching programming to beginners - choosing the language is just the first step. In *Proceedings of the Australasian conference on Computing education*, pages 79–86. ACM Press, 2000.
- [7] H. E. Dunsmore and J. D. Gannon. Programming factors - language features that help explain programming complexity. In *Proceedings of the 1978 annual conference*, pages 554–560. ACM Press, 1978.
- [8] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 1292:369–388, September 1997.
- [9] Stephen N. Freund and Eric S. Roberts. Thetis: an ansi c programming environment designed for introductory use. In *Proceedings of the twenty-*

- seventh SIGCSE technical symposium on Computer science education*, pages 300–304. ACM Press, 1996.
- [10] Johnson R. Gamma E., Helm R. and Vlissides J. *Design Patterns*. Addison-Wesley, 1995.
- [11] J. D. Gannon and J. J. Horning. The impact of language design on the production of reliable software. In *Proceedings of the international conference on Reliable software*, pages 10–22, 1975.
- [12] Cary G. Gray and Michael D. Frazier. Introducing computer science after programming. *The Journal of Computing in Small Colleges*, 18(1):65–76, 2002.
- [13] Martin Hitz and Marcus Hudec. Modula-2 versus c++ as a first programming language: some empirical results. In *Papers of the 26th SIGCSE technical symposium on Computer science education*, pages 317–321. ACM Press, 1995.
- [14] Matthew C. Jadud and Sally A. Fincher. Naïve tools for studying compilation histories. techreport 3-03, University of Kent Canterbury, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, March 2003.
- [15] D. Klahr and S.M. Carver. Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, 20:362–404, 1988.
- [16] Michael Klling and John Rosenberg. Bluea language for teaching object-oriented programming. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 190–194. ACM Press, 1996.
- [17] Michael Klling and John Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36. ACM Press, 2001.
- [18] Joan Krone and Todd Feil. Incorporating mathematics into the first year cs program: a new approach to cs2. *The Journal of Computing in Small Colleges*, 17(1):44–51, 2001.
- [19] J.C. Leslie. *Essential Behaviorism*. Hodder Headline Group, 2002.
- [20] A. Patterson M. Kölling, B. Quig and J. Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education*, 13(4), 2003.
- [21] P. G. Moulton and M. E. Muller. Ditrana compiler emphasizing diagnostics. *Communications of the ACM*, 10(1):45–52, 1967.

- [22] James L. Noyes. A first course in computational science: (why a math book isn't enough). In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 18–22. ACM Press, 2002.
- [23] Seymour Papert. The uses of technology to enhance education. Technical report, MIT, June 1973.
- [24] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, New York, 1980.
- [25] Cuno Pfister. A brief history of pascal, 2003.
- [26] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139. ACM Press, 1997.
- [27] Tom Schorsch. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 168–172. ACM Press, 1995.
- [28] B.F. Skinner. *About Behaviorism*. Knopf, 1974.
- [29] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.
- [30] Elliot Soloway, Kate Ehrlich, and John B. Black. Beyond numbers: Don't ask how many ... ask why. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 240–246. ACM Press, 1983.
- [31] James Spohrer and Eliot Soloway. Analyzing the high-frequency bugs in novice programs. *Empirical Studies of Programmers*, 1986.
- [32] James C. Spohrer, Elliot Soloway, and Edgar Pope. Where the bugs are. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 47–53. ACM Press, 1985.
- [33] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.
- [34] Richard L. Wexelblat. The consequences of one's first programming language. In *Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems*, pages 52–55, 1980.
- [35] Marvin V. Zelkowitz. Automatic program analysis and evaluation. In *Proceedings of the 2nd international conference on Software engineering*, pages 158–163. IEEE Computer Society Press, 1976.