BRUCE L. SHERIN

A COMPARISON OF PROGRAMMING LANGUAGES AND ALGEBRAIC NOTATION AS EXPRESSIVE LANGUAGES FOR PHYSICS

ABSTRACT. The purpose of the present work is to consider some of the implications of replacing, for the purposes of physics instruction, algebraic notation with a programming language. What is novel is that, more than previous work, I take seriously the possibility that a programming language can function as the principle representational system for physics instruction. This means treating programming as potentially having a similar status and performing a similar function to algebraic notation in physics learning. In order to address the implications of replacing the usual notational system with programming, I begin with two informal conjectures: (1) Programming-based representations might be easier for students to understand than equation-based representations, and (2) programming-based representations might privilege a somewhat different "intuitive vocabulary." If the second conjecture is correct, it means that the nature of the understanding associated with programming-physics might be fundamentally different than the understanding associated with algebra-physics.

In order to refine and address these conjectures, I introduce a framework based around two theoretical constructs, what I call *interpretive devices* and *symbolic forms*. A conclusion of this work is that algebra-physics can be characterized as *a physics of balance and equilibrium*, and programming-physics as *a physics of processes and causation*. More generally, this work provides a theoretical and empirical basis for understanding how the use of particular symbol systems affects students' conceptualization.

KEY WORDS: algebra, cognition, physics, programming, representations

1. INTRODUCTION

Almost since the earliest days of computers, it has been proposed that, as part of their educational experiences, students should be asked to program computers. These proposals have taken a variety of forms, have been directed at students of a wide range of ages, and have employed a variety of program languages. For example, there have been proposals to use existing programming languages, such as Fortran, as a way to supplement the instruction of older students (e.g., Bork, 1967). In addition, there have been attempts to develop programming environments specifically for children. These include such environments as Logo (Papert, 1967),

International Journal of Computers for Mathematical Learning **6:** 1–61, 2001.

StarLogo (Resnick, 1994; Wilensky, 1997), Boxer (di Sessa, 2000), and Agentsheets (Repenning, 1993).

In this paper, I will look at the possibility of introducing student programming in one domain: physics. Within the discipline of physics, equations and related mathematical notations play an extremely important role. Physicists write equations and manipulate symbols in order to perform extended and complex computations. Furthermore, physicists use equations as a means of making precise and compact expressions of physical laws, which then appear in academic papers and in textbooks. Moreover, this importance extends to physics classrooms. Students are introduced to fundamental laws through the use of mathematical notations. And much of a physics student's work involves the use of equations to solve problems.

The present work considers some of the implications of replacing, for the purposes of instruction, these algebraic notations with a programming language. What is novel in this work is not the proposal to employ student programming in physics instruction. In fact, we will see that, as a domain, physics has seen some of the earliest and most persistent attempts to apply student programming. Rather, what is novel is that, more than previous work, I will take seriously the possibility that a programming language can function as the principal representational system for physics instruction. This means treating programming as potentially having a similar status and performing a similar function to algebraic notation in physics learning. It means that instead of representing laws as equations, students will represent them as programs.

In considering the implications of replacing algebraic notation with programming, my focus will be narrowed in a number of important respects. Certainly, there would be many superficial – or, at least, obvious – differences in what students would be learning and doing. Students will be learning programming and programming algorithms, rather than some derivational and problem solving strategies that are particular to textbook physics problems. However, my focus will not be on this level of difference. Rather, I am interested in whether there are fundamental differences in learning and understanding that are traceable to features of the particular representational languages employed.

1.1. *Initial Conjectures*

In essence, therefore, the purpose of this work is to compare programming and algebra as 'languages' for physics learning. In the initial stages of this research, I began with two informally stated conjectures concerning important differences in these languages. Ultimately, as we will see, these

hypotheses were refined in a number of significant respects. Nonetheless, it is helpful to begin with a statement of the intuitions that initially guided this work.

1.1.1. *Programs are Easier to Interpret than Equations*

The first informal conjecture was that programs might be easier to understand or *interpret* than equations. This conjecture derives from the observation that programs can be mentally 'run' to generate motion phenomena (Sherin et al., 1993). The idea is that a programmer can do what a computer does and step, line-by-line, through a program, a procedure that appears relatively straightforward. In contrast, when I began this work, I was not aware of any obvious candidates for a universal technique that allows the generation of an associated motion phenomenon from an equation.

This hypothesis, if correct, has important implications for instructional applications. It has been a frequent observation of earlier research that physics students tend to use equations without understanding; they write equations from memory, and manipulate them in a rote manner (e.g., Larkin, McDermott, Simon and Simon, 1980; Sherin, in press). This is seen to have devastating effects on the learning that occurs in physics classrooms, since it means that a student's conceptual understanding of physics is not being positively impacted during the many hours of problem solving that are typical of a physics students' learning activities. Thus, if programs are easier for students to interpret – if students are more likely to use them 'with understanding' – there would be important instructional implications.

1.1.2. *Programming-Physics Privileges a Different Intuitive Vocabulary*

The second informal conjecture is based in the notion that different symbol systems might allow different 'intuitive conceptions' to be more or less explicitly represented, and that those that are directly supported might become privileged in the associated practice of physics. For example, as I will attempt to argue later, there are reasons to believe that programming languages may be better suited for expressing *causal* intuitions.

This second informal hypothesis is of particular importance in the current endeavor because it points to the possibility that the very nature of the understanding engendered by programming-based physics might be different. As it turns out, this suggestion of a deep relationship between conceptual understanding and symbolic formalisms is in agreement with some fundamental intuitions of physicists concerning the nature of physics

understanding. For example, in *The Character of Physical Law*, physicist Richard Feynman states (Feynman, 1965):

> The burden of the lecture is just to emphasize the fact that it is impossible to explain honestly the beauties of the laws of nature in a way that people can feel, without their having some deep understanding in mathematics. I am sorry, but this seems to be the case. (p. 39)
> Why not tell me in words instead of in symbols? Mathematics is just a language, and I want to be able to translate the language. ... But I do not think it is possible, because mathematics is not just another language. Mathematics is a language plus reasoning; it is like a language plus logic. (p. 40)

In these quotes, Feynman expresses the intuition that, in some manner, it is not possible to fully understand physics without being an initiate into the physicist's use of mathematics and the associated symbols. Feynman tells his students that he cannot explain "in words instead of symbols." To understand physics, you must have the experience of living in the tight net of physical relations expressed as equations, going from one to the other. You have to write equations and solve them, derive one equation from another, and from this experience develop a sense for the properties of the equations and the relations among them.

If Feynman is right then, in replacing equations with a programming language, we are not simply substituting one tool for another, we are changing the very nature of physics understanding. This has important implications for how we must understand the comparison that is at the heart of this inquiry. If the second conjecture is correct, it will simply not be appropriate to think of programming and algebra-based instruction as two routes that lead ultimately to the same 'conceptual' understanding. I cannot ask whether programming or algebra does certain jobs better, and I cannot ask which of the systems better 'represents' physics knowledge. As an alternative, I thus propose to ask and answer the following questions: How does the understanding associated with 'programming-physics' differ from 'algebra-physics'? And: Is 'programming-physics' a respectable form of physics?

Other authors have been clear that representations do not simply provide new ways of representing the same knowledge, they also shape and determine the nature of knowledge. For example, in *The Domestication of the Savage Mind*, Jack Goody argues that the development of written lists changed people's practices as well as the very nature of knowledge and understanding (Goody, 1977). Furthermore, similar notions have appeared in many forms, and in a variety of literature. Jerome Bruner spoke of external representations as a type of "cultural amplifier" (Bruner, 1966). Whorf hypothesized that the vocabulary and syntax of a language shape the speaker's view of reality (Whorf, 1956). Lev Vygotsky was

the father of a branch of psychology for which the internalization of external signs plays a fundamental role in the development and character of thought (Vygotsky, 1934/1986). And there is an extensive body of literature devoted to determining the effect of literacy on thought (e.g., Goody and Watt, 1968; Scribner and Cole, 1981).

Although, in a sense, I will be following in the footsteps of these authors, my claims will be much more modest than the strongest versions proposed by earlier researchers. I certainly will not be arguing that the changes in representational systems proposed would lead to sweeping changes in the way that students think, across all domains and contexts. Rather, my claims will be much more localized, while still remaining fundamental; I will be looking for deep differences in *physics* understanding.

1.1.3. *Refining the Hypotheses: Interpretive Devices and Symbolic Forms*
As I mentioned, the informal hypotheses will be refined as I proceed. The refined analysis will be based on a framework that is centered around two theoretical constructs, what I call *interpretive devices* and *symbolic forms*. These two constructs align, roughly, with the two informal hypotheses. This framework has been described in detail elsewhere (Sherin, 1996, in press), and I will describe it in more detail later in this paper. Here, I will just introduce the two constructs briefly and informally.

1. *Interpretive devices.* The conjecture that programs may be interpreted by mentally 'running' them constitutes an hypothesis that interpretations – the natural language utterances that one can make concerning programs – will have a certain structure. The question then becomes: Are there analogous structures associated with equation interpretations? And: Are there other ways that programs are interpreted? I call these structures in interpretations interpretive devices, and the first part of my analysis is to look at the interpretive devices associated with equations and programming.
2. *Symbolic forms.* I will argue that students gradually learn to recognize a certain type of structure in symbolic expressions, and that these structures are associated with elements in a basic conceptual vocabulary. It is this association of symbol structure and meaning that I refer to as a 'symbolic form.' Thus, in terms of symbolic forms, the second hypothesis can now be rephrased as: Are there fundamental differences in the vocabulary of symbolic forms that are associated with algebra-physics and programming-physics?

1.2. *Two Practices for Introductory Physics Instruction*

It is not sufficient to think of programming languages and algebraic notation as possessing properties that, in any simple sense, are solely a function of the literal nature of expressions, as they appear on a piece of paper or a computer display. Rather, to the extent that these representational languages possess 'properties' along our dimensions of interest, these properties will depend on exactly how the symbol systems are used. For this study, I must therefore pick out a particular algebra-physics and a particular programming-physics for study, and it is these practices that I must compare.

Specifying a practice of algebra-physics for study does not pose a significant difficulty. I will examine algebra-physics as it is practiced by students in traditional introductory physics courses in the United States. Because of the great uniformity that exists across such traditional courses (which still make up most physics instruction in the U.S.), this constitutes a relatively well-defined and uniform practice. Perhaps the most prominent aspect of this practice is an emphasis on solving a class of textbook problems that involve the use of equations to find numerical results.

For the case of programming, there is less to build on, and there is certainly no uniformly accepted method of employing programming languages in physics instruction. In this work, I will draw on a practice of programming-physics that was developed by the Boxer Research Group at U.C. Berkeley, and was implemented in 6th grade and high school classrooms (di Sessa, 1989; Sherin et al., 1993). Unlike the problem-based approach typical of traditional introductory physics courses, the courses developed by the Boxer Research Group were structured largely around the programming of computer simulations. For the purposes of this study, this practice was transported into a laboratory setting, and it was employed with university students who had already completed two semesters of introductory physics. The resulting practice will be described in some detail later in the paper.

1.3. *Overview of the Paper*

The preceding sections have been intended to set the stage and frame the inquiry. In what follows, I will begin by discussing prior research. Then, in the sections that follow, my description of the present research program can begin in earnest. In Section 3, I will describe the design of the study, including the participants and tasks, and, in Section 4, I will introduce the analytic framework in the context of a discussion of algebra-physics.

In Section 5, the discussion turns to programming-physics. I will describe the particular practice of programming-physics employed in this

study and, in Section 6, I will apply the analytic framework to that practice. In Section 7, my discussion reaches it apex, and I will draw together my observations in the preceding sections to produce a comparison of algebra-physics and programming-physics along the dimensions captured by the analytic framework. Finally, in Section 8, I will discuss the implications of this work from a broader perspective, and I will conclude the paper.

2. A BRIEF HISTORY OF PROGRAMMING-PHYSICS

The work presented here is not intended, primarily, to argue for programming-physics as a curricular innovation. I will not describe a particular curricular intervention in any detail, nor will I argue more broadly for programming-physics-based instruction. Rather, my purpose is to perform a certain kind of analysis of the understanding associated with programming-physics. Nonetheless, it is still helpful to situate this work within the history of programming-physics as a curricular innovation. This is helpful because, as we will see, I am proposing that we think of programming-physics instruction in a somewhat different manner than has been suggested by prior researchers.

To see the roots of programming-physics instruction, we must go back to a time prior to the use of computers in instruction. Even before the wide appearance of computers in classrooms, some educators had realized the value of a related technique, the use of numerical methods in physics instruction. For example, in his *Lectures on Physics*, Richard Feynman used numerical methods as a key piece of his exposition on forces and $F = ma$ (Feynman et al., 1963). At the heart of Feynman's exposition is the statement that, if we know the starting position and velocity of an object, and if we know the forces on the object as a function of its position and velocity – $F(x,v)$ – then we are in a position to employ a very powerful and general procedure for predicting the motion of an object. As long as you know how to find the force on an object as a function of its position and speed, you can use the expression $a = {}^{F}/_{m}$ to find the acceleration at a given instant, and then plug in and iterate to find the motion of the object at each succeeding instant. The computations may be tedious, but the method will work for every motion.

In his *Lectures*, Feynman went on to apply the numerical method in gory detail for two motions, the oscillation of a mass on a spring, and the motion of a planet around the sun. Then, following his last computation Feynman commented:

> So, as we said, we began this chapter not knowing how to calculate even the motion of a mass on a spring. Now, armed with the tremendous power of Newton's laws, we can not

only calculate such simple motions but also, given only a machine to handle the arithmetic, even the tremendously complex motions of the planets to as high a degree of precision as we wish! (p. 9-9, Volume 1)

As this quote suggests, the reason that Feynman spends time on these laborious computations is that he is trying to convey to his students the fundamental power and generality of Newton's theory. Nevertheless, in Feynman's three-volume text, numerical methods are not to be seen outside of this one exposition. There are certainly many reasons for this, not least of which is that Feynman probably has some commitment to covering the material in its traditional form. However, a major factor must also be the lack of a "machine to handle the arithmetic." Although computers were in use by researchers at the time of Feynman's lectures, they were not available in classrooms.

But computers were to appear on the scene shortly and, with them, the possibility of realizing the benefits of numerical methods. In fact, it wasn't long after the publication of Feynman's lectures that books appeared, some directed at physics teachers, that explained how to use the teaching of programming in physics instruction (e.g., Bork, 1967; Ehrlich, 1973). However, as computers have become more powerful and user-friendly, the move in physics education has been toward uses other than student programming. By the late 1980s, applications such as pre-made simulation environments, computer tutorials, and real-time data-collection tools had already come to dominate. (For a broad picture of work at this time, see Redish and Risley, 1988.)

But student programming in physics was certainly not totally abandoned through the 1980s and 1990s. The Logo community has continued to be active in a number of related domains, including parts of physics (e.g., Wilensky, 1999). Materials are still being produced for teachers, though they are frequently targeted for auxiliary courses in numerical techniques (e.g., Gould and Tobochnik, 1988). And similar approaches are showing up in alternative forms, such as in applications of spreadsheets to physics instruction (e.g., Misner and Cooney, 1991).

A notable example is the M.U.P.P.E.T. project at the University of Maryland, which is an ambitious attempt to integrate Pascal programming into the university physics curriculum (Redish and Wilson, 1993; McDonald et al., 1988). Rather than simply supplementing the standard curriculum with a few programming activities, Redish and colleagues have set out to "rethink the curriculum entirely from the ground up." One of their main points is that limits in students' mathematical abilities, as well as the inherent difficulty of much of the mathematics involved in physics, has placed many constraints on the physics curriculum. By intro-

ducing numerical methods implemented on a computer, many of these constraints are loosened. As Feynman realized, the same, very general, numerical methods can be applied to study a very wide range of physical phenomena. This allows the curriculum to be ordered in a more 'natural' manner, and the range of phenomena that can be studied is greatly broadened.

Where does the current work fit in this brief history? The work I am going to describe is unique because of its tight focus on the representational language itself. Earlier research has been very concerned with what student programming allows. From this point of view, the primary importance of the programming language is that it provides a means of bringing the power of the computer into play. In this view, algebra retains its status as the primary representational system of physics, and programming plays an auxiliary role as the language we use to communicate with computers. In contrast, I am interested in studying programming languages as representational systems in their own right. I want to elevate programming languages to the status of bona fide representational systems for expressing physical laws and relations, and I want to study the properties of these systems. I believe that this move constitutes the next logical step in this history and, when we take this step, we will have moved one notch further toward a true programming-physics.

3. TASKS, DATA CORPUS, AND ANALYTIC METHODS

For the purposes of this work, I created laboratory situations in which to study the two practices under consideration. I chose to study students under a laboratory situation for a number of reasons. Most importantly, I wanted to perform the study under circumstances in which I could capture student work as it developed on a moment-by-moment basis.

All of the subjects in this study were UC Berkeley students enrolled in *Physics 7C*. Physics 7C is a third semester introductory course intended primarily for engineering majors. The fact that these students were in Physics 7C implies that they had already completed two semesters of instruction, Physics 7A and 7B. Many also had some instruction during high school.

The subjects were divided into two distinct pools, an *algebra pool* and a *programming pool*, with each pool consisting of 5 pairs of students. Students in the algebra pool worked with their partner at a whiteboard to solve a pre-specified set of problems. They stood at the whiteboard and, one-at-a-time, I handed them a sheet of paper with a problem to

TABLE I

The time that algebra pool students spent in the experimental sessions

	Mike & Ken	Jack & Jim	Mark & Roger	Jon & Ella	Alan & Bob	Totals
# of sessions	4	5	5	6	6	26
Total hours	5.0	4.5	6.0	6.0	5.5	27.0
Focus hours	2.0	1.5	2.0	3.0	2.5	11.5

solve. Then they solved the problem while I operated a video camera. Occasionally, I interrupted for clarification.

Most of the problems that the algebra pool subjects were asked to solve were fairly traditional textbook problems, but a few more unusual tasks were also included. The pairs typically solved the problems in 5 1/2 hours spread over 5 sessions. The result was a total of 27 hours of videotape of these students using algebraic notation to solve physics problems (refer to Table I). A subset of this data, corresponding to the work of students on 7 of the tasks, was selected for more focused analysis. (These 7 tasks are listed in Appendix B.)

Programming pool subjects worked in pairs at computers. At any time, a single pair of students was in the laboratory, and the students sat together at a single computer, usually with one student typing at the keyboard and the other operating the mouse. As compared with the algebra-physics sessions, there was thus somewhat less of a 'performance' aspect to the sessions, since the students sat facing a computer monitor, rather than standing in front of a whiteboard. Two cameras were positioned quite close to the students, one behind and one to the side. During the sessions, I sat off to the side and operated the cameras, occasionally interrupting for clarification.

The programming pool students were first given about four hours of training, spread over 2 two-hour sessions. Following the training, the students were asked to program a set of simulations of physical motions. They typically required 4 additional two-hour sessions to complete these tasks. In sum, the result was a total of 53 hours of videotape of students programming simulations (refer to Table II). A subset of this data, 16 hours, was selected for more focused analysis. In later sections, I will describe the nature of the programming tasks in much more detail.

One issue concerning the programming pool subjects deserves mention here. It is of great relevance to my examination of programming-physics that the students came to the study with experience in algebra-physics. Because they have this experience, the results described here are almost

TABLE II

The time that programming pool students spent in the experimental sessions

	Adam & Jerry	Ned & David	Amy & Clem	Tim & Steve	Greg & Fred	Totals
# of sessions	6	6	6	6	8	32
Total hours	10.5	9.0	9.5	11.0	13.0	53.0
Focus hours	3.0	3.0	2.0	3.5	4.5	16.0

certainly different than if I had studied students that were introduced to physics solely through a programming-based practice.

The reasons for choosing to study experienced physics students in the programming portion of the study were largely pragmatic. It would simply have been too time consuming to create, from scratch, a population of true programming-physics initiates. Furthermore, there are some reasons that this approach is not too problematic. First, as I will describe below, my analysis was always tightly focused around the external representations; I only looked at what people said while actually looking at and pointing to particular programming expressions. This means that my analysis is only revealing where and how understanding gets connected to programming expressions; it is not giving a full picture of my students' physical intuition or other physics-related knowledge. Of course, because these students have significant experience in seeing structure in algebraic expressions, there is likely still some skewing of my programming-physics results in the direction of algebra-physics. But this just means that any differences uncovered here are likely, if anything, to underestimate the differences that would be revealed by a study of true programming-physics initiates.

All of the videotaped sessions selected for focused analysis were carefully transcribed. The framework and results presented here are based on a systematic analysis of the resulting corpus of transcripts. However, in this paper, I will not describe the nature of this systematic analysis in any detail. Such a description would add greatly to the length of this paper and, I believe, it would not significantly contribute to my core arguments. Instead, I will support my arguments by presenting a selection of examples from the corpus, and using them to support the plausibility of my account. (For more detail on the nature of this systematic analysis refer to Sherin, 1996; Sherin, in press.)

Here, I will just say a little more about how the analysis proceeded. The analysis, of both parts of the data corpus, began by identifying a particular class of episodes, episodes in which symbolic expressions were used 'with understanding.' Roughly speaking, this boiled down to two types of

episodes: (1) episodes in which a student interpreted (stated the 'meaning' of) an expression, and (2) episodes in which a student constructed an expression from some understanding of what they wanted to express. These episodes were selected because, it was presumed, this would allow me to see what type of meaning is associated with the symbolic expressions in each practice, and how it gets associated. In the following sections, I will present examples of this sort in order to support my account.

Once these episodes were identified, I iteratively viewed and reviewed all of the transcripts corresponding to the focus tasks, and I applied the framework to these episodes. After each of these iterations, the framework was revised.

4. THE ANALYTIC FRAMEWORK INTRODUCED AND APPLIED TO ALGEBRA-PHYSICS

In this section, I will undertake two goals. First, I will introduce the framework – namely, symbolic forms and interpretive devices – in a more concrete manner. Second, I will apply the framework to algebra-physics. Throughout this section, my purpose will only be to describe and illustrate the framework. I will not attempt to argue for it, or for my interpretation of individual episodes. For arguments of that sort, the reader is referred to Sherin (in press).

In addition, I will not discuss, in any detail, the relationship of this framework to existing literature. To construct my analytic framework, I have built on – and, I hope, contributed to – some prior work by mathematics education researchers. For example, I have relied significantly on research into how students understand and solve elementary word problems (e.g., Carpenter and Moser, 1983; Fuson, 1992; Reed, 1998; Riley, Greeno and Heller, 1983) and understand mathematical equations (Kieran, 1992; Herscovics and Kieran, 1980; Sfard, 1987, 1991). In addition, I have drawn on much work from physics education research. Most centrally, I have built upon di Sessa's work concerning p-prims and the sense-of-mechanism (di Sessa, 1993). To a lesser extent, I have also drawn on qualitative physics research (deKleer, 1984; Forbus, 1984; Hayes, 1979, 1984) and on work concerning physics problem solving (Chi et al., 1981; Larkin, 1983; Larkin et al., 1980). A reader familiar with this prior work may recognize these influences as my exposition proceeds, but I will not elaborate on these connections further. Here, again, the interested reader is referred to Sherin (in press) for more discussion.

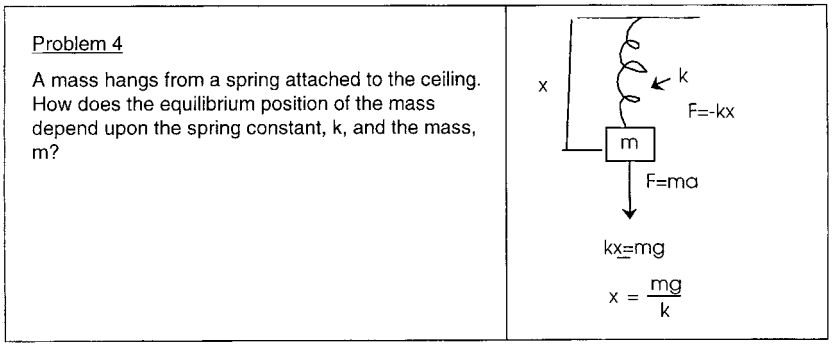


Figure 1. Jack and Jim's solution to the spring problem.

4.1. *A First Example*

It is easiest to understand the framework by grounding the discussion in specific example episodes. As I stated briefly above, I am interested in episodes in which a student associates 'meaning' with an equation. In this first episode, I will present a simple example in which a student does just that; he points to an equation and says what it means.

This first episode is taken from the work of a pair of students, Jack and Jim, on Problem #4. In this problem, a mass hangs motionless at the end of a spring. The problem, along with Jack and Jim's solution, is reproduced in Figure 1. Looking at this figure, we can see that, as their final result, Jack and Jim derived the expression:

$$x = \frac{mg}{k}$$

where x is the amount that the spring stretches, m is the mass, k is the spring constant, and g is the acceleration due to gravity. After this expression was derived, Jim spontaneously explained that this is a completely sensible result:[1]

Jim: Okay, and this makes sort of sense because you figure that, as you have a more massive block hanging from the spring, then your position x is gonna increase, which is what this is showing. [Points to m then k.] And that if you have a stiffer spring, then your position x is gonna decrease. [Uses fingers to indicate the gap between the mass and ceiling.] That's why it's in the denominator. So, the answer makes sense.

Jim says that this expression makes sense for two reasons. First, he tells us that the equation says that, if you increase the mass (with k held constant) then the spring stretches farther and x increases. Second he considers the case where the spring is made stiffer; that is, k is increased. In this case, he says, the equation implies that x will decrease. Presumably, these observa-

tions make sense to Jim because of knowledge that he has concerning the behavior of springs.

4.1.1. *Interpretive Devices Introduced*

I consider the above statement by Jim to be an *interpretive utterance*, or, more simply, an *interpretation*. Jim is not merely providing us with a literal reading of the symbols on the page; he does not simply say "X equals M G over K." Rather, he tells us what the equation 'says.' For example, one thing that the equation says is that if you increase k then x decreases.

This is the sort of phenomena with which this research is concerned. In the introduction to this paper, I speculated that expressions in programming-physics could be interpreted by mentally 'running' them. However, I left open the question of whether there would be analogous interpretations in algebra-physics. In Jim's statement above, we have found at least one kind of algebra-physics utterance that is analogous to mental running. With that in mind, how can we characterize Jim's interpretation?

Even this very simple interpretation has a number of interesting characteristics. In saying that as k increases x decreases, Jim is imagining a very particular process: k is increasing rather than decreasing, and m and g are parameters that are implicitly held fixed. Similarly, when Jim says "as you have a more massive block," he is imagining a process in which the mass increases, but other parameters are held fixed.

Thus, Jim's interpretive utterances in the above passage have a very particular structure: he imagines that a quantity varies while other quantities are held fixed. One might now, reasonably, begin to wonder whether all interpretations in algebra-physics have this structure. Although it turns out that this form of interpretation is not uncommon, there were also many other types of interpretations. As part of my analysis of the data corpus, I looked at the interpretive utterances made by students, and put them into categories according to structures of roughly this sort. This resulted in a number of categories, each of which is associated with what I call an *interpretive device*. An overview of the full list of interpretive devices will be given in a later section.

CHANGING PARAMETERS	A narrative interpretation in which one quantity is varied while others are held fixed.

In the case of interpretations like Jim's – where one imagines that one quantity is varied while others are held fixed – I call the associated interpretive device CHANGING PARAMETERS. This interpretive device is part of a larger class that I call *Narrative* devices. As a whole, I call these devices 'narrative' because they often lead to interpretive utterances that

are like a very short story or short argument. The use of the term 'device' is intended to suggest that interpretive devices are like the literary devices for the very brief stories in narrative interpretations.

4.1.2. *Symbolic Forms Introduced*

The preceding discussion of Jim's interpretation leaves open a number of essential questions. I characterized the structure of Jim's interpretive utterance, and I looked at what Jim said that his equation meant. But, how did Jim know that this is the meaning of this particular equation? What exactly are the features of the equation that he is attending to, and that tell him what the meaning of the equation is?

The analysis associated with symbolic forms is, in part, intended to answer these questions. According to the hypothesis associated with symbolic forms, students learn to recognize a certain type of structure in symbolic equations, and this supports the interpretation and construction of equations. More specifically, each symbolic form involves an association between two components, a *symbol template* and a *conceptual schema*.

- *Symbol template.* The symbol template specifies the structure of an expression at a particular level of detail. For example, the symbol template $\square + \square + \square \ldots$ specifies an expression in which two or more terms are separated by plus signs. Similarly, the symbol template $\square = \square$ specifies an expression in which two sub-expressions are separated by an equal sign.
- *Conceptual schema.* The conceptual schemata associated with forms specify simple relationships among a small number of entities. These schemata are similar in nature to di Sessa's p-prims (1993) and Johnson's (1987) image schemata.

Stated informally, the conceptual schema specifies an idea, and the symbol template specifies how to write that idea in a symbolic expression.

The symbolic forms underlying Jim's interpretation provide some first, relatively simple, examples. There are two symbolic forms involved here, what I call PROPORTIONALITY PLUS (PROP+) and PROPORTIONALITY MINUS (PROP−).

Schema	Symbol template
PROP+	
One quantity varies directly with a second quantity.	$\left[\frac{\ldots x \ldots}{\ldots}\right]$
PROP−	
One quantity varies inversely with a second quantity.	$\left[\frac{\ldots}{\ldots x \ldots}\right]$

The schema for PROP+ has two entities: one quantity is seen as directly proportional to a second quantity. In the case of Jim's interpretation, the position, x, is seen as directly proportional to the mass of the block. Furthermore, the symbol template for PROP+ specifies a very simple symbol structure; when a student sees an expression in terms of PROP+, they are attending to the simple fact that a certain quantity appears in the numerator. In Jim's interpretation above, he is attending to the fact that the mass, m, is in the numerator of the expression.

A similar analysis holds for the second part of Jim's interpretation. In this second part, he is seeing the expression in terms of the PROP− form. He attends to the fact that the spring constant, k, is in the denominator, and he concludes that the position will be indirectly proportional to k: "if you have a stiffer spring, then your position x is gonna decrease. That's why it's in the denominator."

In the case of this particular example, the analysis in terms of symbolic forms may seem to boil down to something very simple. Jim is able to say "if you have a stiffer spring, then your position x is gonna decrease" simply because he sees that a k is in the denominator. But the forms analysis is less trivial than this example might suggest. Note that, in presenting a list of symbolic forms – as I will do in the next sub-section – I am essentially hypothesizing that very particular structures are meaningful to students, and that these structures are associated with particular meanings. Thus, when I present an exhaustive list of symbolic forms for algebra-physics, I will essentially be giving a complete picture of the conceptual vocabulary supported by expressions in algebra-physics.

4.2. *Devices and Forms, in Overview*

The full list of the interpretive devices necessary to account for the observations in my data corpus are given in Table III. Similarly, the full list of symbolic forms is given in Table IV. These lists are based on the systematic analysis of the data corpus. For example, as part of that analysis, I identified all of the interpretive utterances in the data corpus, and put them into the categories that were ultimately associated with interpretive devices. As I stated above, this was accomplished in an iterative fashion; I made three passes through all of the transcripts, coding and recoding the interpretive utterances and, on each iteration, the categories were refined. One reason that I undertook this systematic analysis was to get an idea of the frequency with which the various forms and devices appeared. Though I may comment occasionally on the results of this frequency analysis, as discussed above, I will not present the procedure in detail. (Again, see Sherin, 1996 for details.)

TABLE III

Interpretive devices by class

Narrative	Static
CHANGING PARAMETERS	SPECIFIC MOMENT
PHYSICAL CHANGE	GENERIC MOMENT
CHANGING SITUATION	STEADY STATE
	STATIC FORCES
Special case	CONSERVATION
RESTRICTED VALUE	ACCOUNTING
SPECIFIC VALUE	
LIMITING CASE	
RELATIVE VALUES	

TABLE IV

Symbolic forms by cluster

Competing Terms cluster		**Terms are Amounts cluster**	
COMPETING TERMS	$\square \pm \square \pm \square \ldots$	PARTS-OF-A-WHOLE	$[\square + \square + \square \ldots]$
OPPOSITION	$\square - \square$	BASE ± CHANGE	$[\square \pm \Delta]$
BALANCING	$\square = \square$	WHOLE − PART	$[\square - \square]$
CANCELING	$0 = \square - \square$	SAME AMOUNT	$\square = \square$
Dependence cluster		**Coefficient cluster**	
DEPENDENCE	$[\ldots x \ldots]$	COEFFICIENT	$[x\square]$
NO DEPENDENCE	$[\ldots]$	SCALING	$[n\square]$
SOLE DEPENDENCE	$[\ldots x \ldots]$	**Other**	
Multiplication cluster		IDENTITY	$x = \ldots$
INTENSIVE·EXTENSIVE	$x \times y$	DYING AWAY	$[e^{-x\cdots}]$
EXTENSIVE·EXTENSIVE	$x \times y$		
Proportionality cluster			
PROP+	$\left[\frac{\ldots x \ldots}{\ldots}\right]$	RATIO	$\left[\frac{x}{y}\right]$
PROP−	$\left[\frac{\ldots}{\ldots x \ldots}\right]$	CANCELING(B)	$\left[\frac{\ldots x \ldots}{\ldots x \ldots}\right]$

In Table III, the interpretive devices are grouped into three *classes*: Narrative, Static, and Special Case. I will just say a little about each of these classes:

– *Narrative class.* Representational devices in the Narrative Class embed an equation in an imaginary process in which some type of changes occur. The CHANGING PARAMETERS device, which we encountered above, is a device in this class.

- *Static class.* In contrast to Narrative devices, devices in the Static Class project an equation into a static situation. Rather than playing a role in a story, the equation is treated as a description or snapshot of a single moment in a motion. In the next section, I will present example episodes to illustrate some of these devices.
- *Special Case class.* For interpretations based on Narrative and Static devices, the statements made are presumed to be true for any values of the quantities that appear in the expression. For example, when Jim states that when m increases, x increases, this observation is assumed to be true independent of the values of the quantities involved. In contrast, in Special Case devices, conclusions are drawn for cases in which the values of quantities that appear in an expression are somehow restricted. For example, in a LIMITING CASE interpretation, a student imagines what happen when a quantity takes on a particular extreme value, such as zero.

The full list of symbolic forms is given in Table IV, grouped into categories called 'clusters.' This table is extremely important because, in essence, it lays out the conceptual vocabulary that is supported by expressions in algebra-physics.

- *Competing Terms.* In forms in the Competing Terms cluster, terms in a symbolic expression are associated with influences in competition. These influences may be forces (in the technical sense), but they may also be other sorts of quantities, such as momenta. I will illustrate some of the forms in this cluster in the next section.
- *Dependence.* The forms in this cluster have to do with whether or not a specific individual symbol appears in an expression. Most basic of these forms is NO DEPENDENCE, whose symbol pattern involves the absence, rather than the presence, of symbols. In contrast, the NO DEPENDENCE form specifies only that a given symbol appears in an expression. I will present examples to illustrate these forms in the next section.
- *Proportionality.* When a physics student looks at an equation, the line that divides the top from the bottom of a ratio is a major landmark. Forms in the Proportionality Cluster involve the seeing of individual symbols as either above or below this important landmark. The PROP– and PROP+ forms, discussed above, are elements of this cluster.
- *Terms are Amounts.* Like the forms in the Competing Terms Cluster, these forms address expressions at the level of terms. However, rather than describing a battle between competing influences, these expressions concern the collecting of a generic substance. Each term is seen as either contributing to or subtracting from the pool of substance.

- *Coefficient.* Forms in the coefficient cluster break a product of factors into two parts. The first part, usually written on the left, is the coefficient.
- *Multiplication.* The forms in this cluster also break down products of factors into parts. In this case, the parts are intensive or extensive quantities.

4.3. *Further Examples from Algebra-Physics*

In this section, I will present several more example episodes in order to set up the comparison of algebra-physics and programming-physics that will be presented later in the paper. The episodes I will present here all involve students' work on Problem #3, the Air Resistance problem (refer to Appendix B). In this problem, the students are asked to consider a situation in which a ball is dropped from a great height. Because of the force from gravity pulling downward, the ball initially speeds up. As it goes faster, the force of air resistance, which tends to impede the motion, grows larger. Eventually the force from air resistance becomes large enough that it is equal to the force from gravity. After this time, the ball's speed remains at a constant value that is known as the *terminal velocity*.

4.3.1. *Associating an Equation with a Moment*

As a first step toward solving the Air Resistance problem, some students jumped directly to the time when the forces from air resistance and gravity are equal, and terminal velocity has been reached. Mark and Roger were one pair that began in this manner. They wrote the expression $R = mg$, where R is the force from air resistance and mg is the force from gravity. After they wrote this expression, they made some comments that I take to be an interpretation:

$$R = mg$$

Mark: That's when the forces are equal then, right?
Roger: Yeah, because then you wouldn't have acceleration anymore.
Mark: Right. When the force is zero the acceleration's zero.
Roger: Okay. I guess. Okay. After a certain time.
Mark: R equals G.
Roger: At T, some T.

Here, Mark and Roger are telling us something about this equation, but it is something very different than Jim's interpretation above. It does not seem appropriate to describe Mark and Roger's interpretation in terms of quantities that are varied and quantities that are held fixed. Instead, they

are associating the equation with a particular moment in the motion, the moment at which terminal velocity is attained.[2]

SPECIFIC MOMENT	An equation is treated as a description of a particular instant.

I call the interpretive device here SPECIFIC MOMENT. In a SPECIFIC MOMENT interpretation, an equation is essentially taken as a description of a specific instant in the motion under study. Ella, working with her partner Jon, also wrote an expression of this form, and gave a SPECIFIC MOMENT interpretation.

$$Cv^2 = mg$$

Ella: The terminal velocity is just when the – I guess the kind of frictional force from the air resistance equals the gravitational force?

There is also a new symbolic form to discuss here. I believe that students understood these initial equations, which equated the forces of gravity and air resistance, in terms of a symbolic form that I call BALANCING.[3] In the conceptual schema associated with BALANCING, two competing influences are seen as equal and opposite. In the symbol template, each of these influences are associated with expressions that are separated by an equal sign.

BALANCING

Two competing influences are precisely in balance; they are seen as equal and opposite.	□ = □

4.3.2. *Writing an Equation that Describes the Whole Motion*

Looking at another pair of students, Mike and Karl, will allow me to illustrate some additional forms and devices. Unlike the above students, Mike and Karl did not start by writing an expression to describe the situation at terminal velocity. Instead, they began by writing a more general expression that describes the entirety of the motion, including the time when the ball is still speeding up. They drew the diagram shown in Figure 2, and then they composed the following equation, talking as they wrote:

$$a(t) = -g + \frac{f(v)}{m}$$

Mike: So, at least we can agree on this and we can start our problem from this scenario. [Indicates the diagram with a sweeping gesture.] Right? Okay? So, at any time,, At any

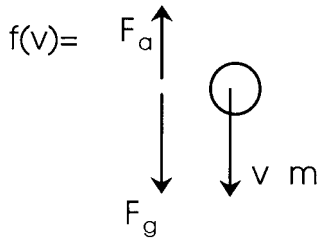


Figure 2. Mike and Karl's air resistance diagram.

time, the acceleration due to gravity is G, and the acceleration due to the resistance force is F of V over M. [Writes "g + f(v)/m".]
Karl: Ah :::
Mike: Okay, now they're opposing so it's a minus. So, this is negative G. [Motions downward, writes a negative sign in front of the g.] Positive direction. You have negative G plus F of V over M. That's you're acceleration at any time. Right?
Karl: Well, wait. You want them to have.
Mike: This is the acceleration at any time T. [Writes "a(t)" next to the expression.]

In this last passage, Mike and Karl's conceptualization of the situation takes shape as an equation. Mike first looks at the diagram, saying that this is the "scenario" that they will start with. Then he says that the acceleration is due to two factors, force and air resistance, and he writes $g + \frac{f(v)}{m}$. Then he notes that, because these two factors are "opposing," there must be a minus sign in front of the gravity term.

I believe that Mike and Karl are understanding the situation in terms of two influences that are competing to produce a result, and they have directly embodied this understanding in an equation. In terms of symbolic forms, I say that three specific symbolic forms are involved in the construction of this expression: COMPETING TERMS, OPPOSITION, and IDENTITY.

COMPETING TERMS	$\square \pm \square \pm \ldots$
OPPOSITION	$\square - \square$
IDENTITY	$x = [\ldots]$

COMPETING TERMS. Like the BALANCING form, the COMPETING TERMS form involves influences in competition. In this case, the two influences are gravity and air resistance, and each of these influences is associated with one of the terms in the equation.
OPPOSITION. This is essentially a special case of COMPETING TERMS. In the OPPOSITION form, two influences are competing in a specific manner: they are opposing. The symbol template specifies that a minus $(-)$ sign will appear between the two terms associated with these influences. As Mike says: "now they're opposing so it's a minus."
IDENTITY. In the IDENTITY form, a single symbol, usually written on the left, is set equal to an expression. This is the form that allows Mike to write $a(t) =$ on the left side of his expression. As he writes this, he says: "This is the acceleration at any time T."

Mike and Karl's work can also be used to illustrate a new interpretive device. Note that, unlike the earlier BALANCING expressions, Mike and Karl's expression is not true only when terminal velocity is reached. As Mike says in his interpretation, their expression gives the acceleration "at any time;" rather than describing a particular moment, it describes any moment in the motion. For this reason, I call the interpretive device that is active here GENERIC MOMENT.

4.3.3. *Composing an Expression for Air Resistance*

Before concluding my discussion of the Air Resistance problem, I want to discuss one final set of examples. It turns out that writing an expression for the force due to air resistance posed a problem for the students. Although some of the students may have seen expressions for this force earlier in their training, none of them appeared to remember these expressions. Thus, they were placed in the situation of having to construct (or reconstruct) an expression for the air resistance force. In all cases, the students began by noting that the force from air resistance must depend on velocity. For example, Mike wrote the expression $F_{air} = f(v)$ and commented:

Mike: We also know that force of air resistance is a function of velocity. Right?

Then this statement was refined to say that the force should get greater as the velocity increases.

$$F_{air} = kv$$

Bob: Okay, and it gets f-, it gets greater as the velocity increases because it's hitting more atoms of air.

There are two symbolic forms involved in the construction of these expressions for the force of air resistance. First, when Mike and Karl write $F_{air} = f(v)$ this is due to the DEPENDENCE form.

DEPENDENCE [. . .*x*. . .]

The DEPENDENCE form is, in a sense, the simplest of all forms. The symbol pattern specifies only that a particular symbol will appear, in some manner, in an expression, and the conceptual schema specifies that one quantity will depend on another. When Mike invoked dependence in the above statement, he was stating simply that the symbol 'v' must appear somewhere in the expression. Other students also began by announcing this dependence:

Mark: So this has to depend on velocity. That's all I'm saying. Your resistance – the resistor force depends on the velocity of the object.

But DEPENDENCE only gets the students so far; it only tells them that the 'v' must appear somewhere. In later steps, the students had to refine their specification for this expression. Bob's statement above does this. He doesn't only say that F_{air} will depend on the velocity, he says that it will increase with increasing velocity, thus implicating the PROP+ form, which we have encountered before.[4]

In addition to DEPENDENCE and PROP+, two other symbolic forms are involved in the construction of the various expressions for the force due to air resistance. First, the IDENTITY form, mentioned above, specifies the overall structure of the expressions as $x = [\ldots]$. And, the COEFFICIENT form specifies that a single symbol, corresponding to a constant parameter, will multiply the expression on the right hand side.

Finally, I want to mention that a new and important interpretive device is involved in Bob's statement that F_{air} "gets greater as the velocity increases." This is a variety of Narrative interpretation; Bob is describing what happens as one quantity changes. However, the device involved here differs in a noteworthy respect from the CHANGING PARAMETERS device. Notice that, in Jim's CHANGING PARAMETERS interpretation above, the change that he described bore no relation to any real motion. He described a process in which the mass changed; but this is a truly imaginary process, it does not correspond to any real motion of a physical system. In contrast, when Bob says "it gets greater as the velocity increases," he is talking about what happens in the motion under consideration. I call the interpretive device associated with this variety of interpretation PHYSICAL CHANGE.

In some cases, PHYSICAL CHANGE interpretations are more dramatically framed as telling the story of a motion. For example, as part of their solution, Alan and Bob wrote the equation $\Sigma F = mg - kv$ for the total force on the falling object. Then Alan gave a physical change interpretation:

$$\Sigma F = mg - kv$$

Alan: And so I guess we were just assuming that ... we would have gravity accelerating the ball and then as, as we got a certain velocity, [points to 'kv'] this would contribute to making accelerating closer – de-accelerate it and so we would just have a function of G minus K V. And then we assumed that since we know that it reaches terminal velocity, its acceleration's gonna be zero, we just assumed G minus K V would equal zero.

In this interpretation, Alan has wrapped the whole story of the motion around the equation.

The existence of physical change interpretations helps to highlight some dramatic features of the earlier interpretations that I presented. In the introduction, I speculated that students might interpret programs by mentally running them. Such an interpretation would essentially tell the

story of the motion that is simulated, from beginning to end. We might have expected a similar circumstance in algebra-physics; we might have expected that all interpretations would be PHYSICAL CHANGE interpretations and would somehow tell the story of a motion around the equation. However, what we have seen is that physical change interpretations were only one of the types of interpretations.

5. PROGRAMMING-PHYSICS AND BOXER

5.1. *A Practice of Programming-Physics*

My discussion now turns to programming-physics, and I will begin by introducing the reader to the practice of programming-physics that was employed in this study. All student programming in this study was done in a new and rather unique programming environment known as Boxer, which employs its own programming language (di Sessa and Abelson, 1986; di Sessa et al., 1991). The Boxer programming environment and language are currently under development at U. C. Berkeley. In many respects, Boxer can be understood to be a direct descendent of Logo, and it has been designed with the specific aim of correcting some of the shortcomings of Logo while maintaining many of the same goals and strengths (di Sessa, 1986).

As I mentioned in the introduction, the particular practice of programming-physics that I used was adapted from work done by the Boxer Research Group. In that research, programming was used as part of physics courses that the Boxer Research Group designed for 6th graders and high school students (di Sessa, 1989; Sherin et al., 1993). To illustrate this practice, I will present a few examples from the brief curriculum given to the college students in this study. In one of the earliest tasks given to the students, they were simply told to "make a realistic simulation of the motion of a ball that is dropped from someone's hand." In response, they created programs like that shown in Figure 3 which was made by two students in this study, Anne and Clem.

Figure 3 shows two types of boxes, a graphics box containing a sprite – the sprite, in this case, is shaped like a ball – and a do-it box named **drop**, which contains the program itself. Anne and Clem's program begins by executing a **setup** command, which positions the sprite at the top of the graphics box and orients it so it is prepared to move downward. Each of the remaining lines in the program contains a **fd** command followed by the **dot** command. The **fd** command instructs the sprite to move forward the specified number of steps, and the **dot** command causes the sprite to

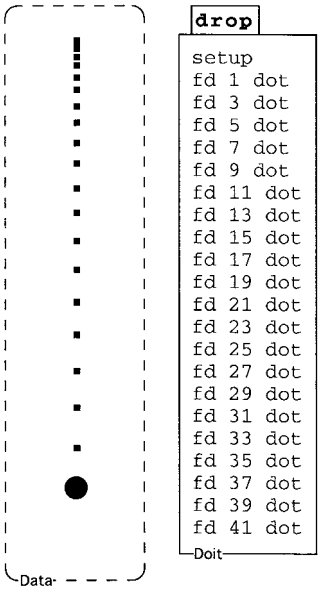


Figure 3. Anne and Clem's simulation of a dropped ball without variables.

draw a dot at its current location. Thus we see that this program causes the sprite to move forward in larger and larger steps, leaving a trail of dots in its wake.

It is important to note how this sort of program was understood by students. All of the pairs tended to assume, at least implicitly, that each of the movements of the sprite takes the same amount of time, an assumption that was reinforced by later aspects of the curriculum. Given this assumption, Anne and Clem's program suggests that the ball is speeding up.

To this point in the curriculum, Anne and Clem had only been introduced to the **fd** and **dot** commands; they had not yet been told how to use variables in Boxer, or how to create programs that iterate. Of course, creating simulations in this manner, using only a sequence of forward commands, would be quite tedious for students and not very illuminating. So, after making only a few programs along the lines of Figure 3 the students were taught to use variables and to apply them to create simulations. In particular, they were taught to use a very specific structure, called the "Tick Model", to program their simulations.

Figure 4 shows Tim and Steve's simulation of the dropped ball, written using the Tick Model. Programs written with the Tick Model use three variables, **pos**, **vel**, and **acc**, which correspond to the position, velocity and acceleration of the object. Furthermore, each step in the motion is handled

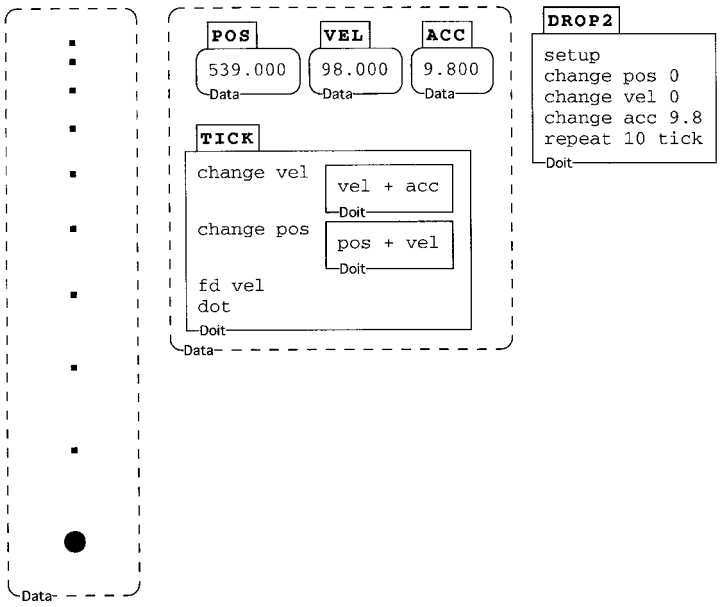


Figure 4. Tim and Steve's simulation of a dropped ball using the Tick Model.

by a procedure named **tick**. Each time that this procedure is executed, the value of **acc** is added onto **vel**, **vel** is added onto **pos**, and the sprite is told to move forward an amount equal to **vel**. Notice that, when the drop procedure is rewritten to employ the **tick** procedure, it becomes somewhat simpler. This new version of **drop** just initializes all of the variables, then it repeats the **tick** procedure 10 times. Once the Tick Model was introduced, students were directed to use it in all of their simulations.

As in this example, the goal of each task was the production of a simulation of some particular motion. For example, one task in my study gave students only the following instructions:

> In this task, you must imagine that the block shown is attached to a spring so that it oscillates back and forth across the surface of a frictionless table. Make a simulation of this motion.

As in this example, instructions were typically very brief; they usually just described a motion in simple terms and then asked students to create a simulation of that motion. (Refer to the list of tasks in Appendix C.)

5.2. *Tim and Steve: Simulating the Drop with Air Resistance*

In this section, I want to describe a more interesting and extended programming episode. This will help to further provide the reader with a feel for

the practice of programming-physics that is under study, and will set the stage for later analysis.

In Task #4, Tim and Steve were asked to modify their simulation of the dropped ball so that it included air resistance. Recall that the algebra-physics subjects were also given a task that involved air resistance. As part of that task, the students were faced with the need to invent their own expressions to account for the resistive effect of the air. Most commonly, they invented a relation for the force due to air resistance of the form $F_{air} = kv$.

The programming task placed students in a similar predicament. Tim and Steve knew that the force acting down was equal to mg – the force of gravity – but, like the algebra-physics students, they did not immediately know how to find the force of air resistance. In the following discussion, this issue is worked out:

Tim: Anyway. Well, when you first start out there's no air resistance. I mean there's no – but as you go faster the little V – little vector here starts growing. [Points to arrow at top of diagram.] . . . So we have to have the size is bigger if you're moving faster. (. . .)
Steve: I don't think we really have to worry about that. It just decreases.
Tim: The faster you're moving the bigger the - so what is this. So this is – this is something times velocity. . . .
Steve: Yeah, okay. So it's a function of velocity, right?
Tim: Yeah.
Steve: Okay. So what should we do? Just make it like one times – point one times V or something?

The above discussion is very similar to discussions in which the algebra-physics students constructed equations for the force of air resistance. Like their predecessors in this study, Tim and Steve came to the conclusion that the force of air resistance should be a constant times the velocity of the falling object. With this decided, they created a variable named res and modified their program to compute a value for this variable, as shown in Figure 5.

In its present state, Tim and Steve's program computes a value for the variable **res**, but this variable does not affect any other aspects of the simulation. Thus, the students still needed to **modify** the tick procedure so that the size of the air resistance would somehow be taken into account in the acceleration. Tim and Steve were aware of this, and they knew that they wanted a statement that began with "**change acc** . . ." inside the **tick** procedure. However, as they set out to write this line, they realized that it wasn't clear to them exactly where to place it among the other lines in **tick**. They wondered: Should we change the acceleration before or after changing the velocity?

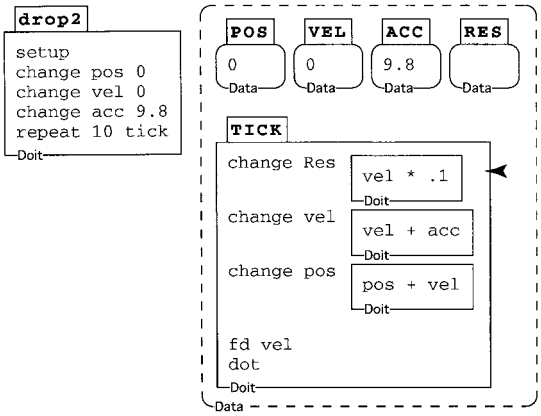


Figure 5. A line is added to compute air resistance.

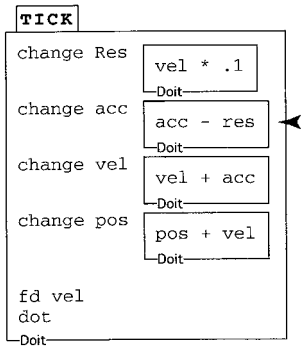


Figure 6. A line is added to compute acceleration.

Tim: No we have to (then account) in the acceleration. So we have to change the acceleration. Do we change the acceleration before we change the velocity? [Pointing into **tick** box here.] Or do we change acceleration after we change velocity? Which also changes the resistance. Oh, I'm getting confused. What's the order in which you change things? (3.0) You see what I'm sayin'? They all like change each other.
Tim: So it matters which order you do it in, right?
Steve: I think the velocity's first, right? . . .
Tim: Well you have to change resistance because you change the velocity, because (0.5) the velocity (1.0) – the new velocity is dependent upon the acceleration which is dependent upon the resistance. So we should change resistance, then acceleration, then velocity. Right? Does that make sense, or am I just talking nonsense.

In the above discussion, Tim and Steve decided on a specific ordering for the various lines that must appear in the **tick** procedure: **res** would be changed first, then **acc**, then **vel**, then **pos**. They next modified **tick** to reflect this decision. This required only the addition of a single new line to compute the value of the acceleration, as shown in Figure 6.

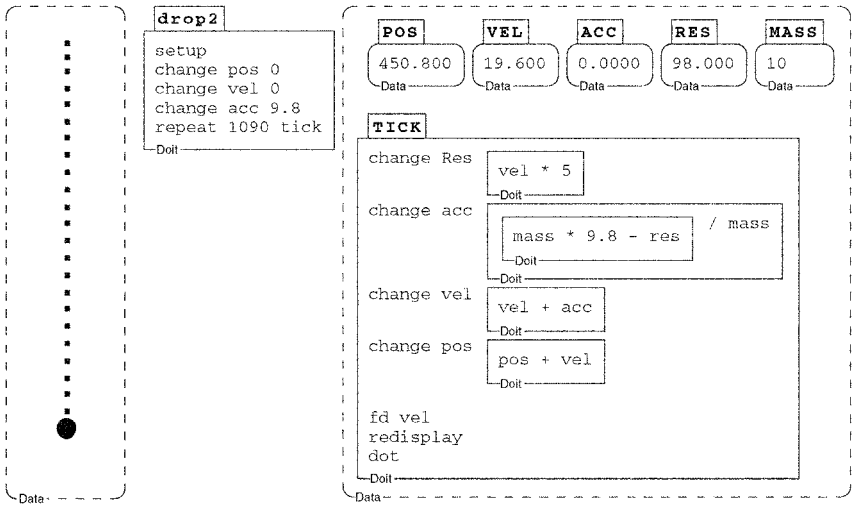
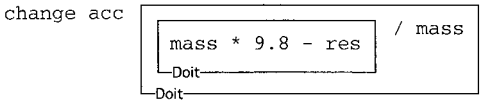


Figure 7. Tim and Steve's final air resistance program.

This program is nearly complete, but the line that computes the acceleration is problematic. In fact, with the program in its current form, the simulation produces a striking behavior: the ball oscillates up and down. It took quite a while for Tim and Steve to correct this problem and ultimately get a working simulation. Eventually, they modified their acceleration line to be:



If rendered in algebraic notation, this statement can be written as:

$$a = \frac{mg - F_{air}}{m}$$

When this modification was made, the program appeared as shown in Figure 7. Tim and Steve ran this program, and were pleased with the results. They could tell that their program was working properly by watching how the values of certain variables changed during the simulation. In particular, they noticed that the acceleration died away to zero and that the velocity leveled off at a constant value, thus indicating that a constant velocity had been reached.

6. THE FRAMEWORK APPLIED TO PROGRAMMING

We are now ready to begin the project of applying the form-device framework to programming-physics. The first step will be to identify the places where programs are constructed and interpreted much like algebraic expressions, and to identify where construction and interpretation are very different in programming. When this is done, we can focus our attention on the places of divergence. In fact, I will argue that there is both substantial overlap and interesting divergence.

To begin this discussion, I need to first take a moment to reflect on the types of meaningful structure that we observed in algebra-physics. Although this point was not emphasized, my description of algebra-physics always focused on individual equations or portions of individual equations. This implicitly presumed that, for my subjects, there were not symbolic forms that span equations – a position that I arrived at after some analysis of my data corpus. In contrast, I will claim that, in programming-physics there are symbolic forms that span multiple lines of programming. Then, what I hope to show is that it is in *within-line* structure that a great overlap between programming and algebra appears. And the locus of substantial divergence will be the *line-spanning* structure in programming.

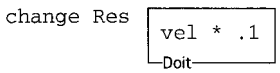
The claim that there are no line-spanning symbolic forms in algebra-physics is one which readers may find controversial. Although students certainly do associate meaning with the arrangement of algebraic expressions – and thus with line-spanning structure – this is not the same sort of meaningful structure that is defined to be involved with symbolic forms. Symbolic forms allow a person to conceptualize *a physical circumstance* in terms of a conceptual schema, and then reflect that conceptualization in an arrangement of symbols. In contrast, the arrangement of algebraic situations typically reflects the *solution process*, rather than an understanding of the physical circumstance.

6.1. *Where Programs Can Be Interpreted Like Equations*

The point of the above discussion was to warn the reader that, in programming, line-spanning structure is going to be important. While the entirety of a program or procedure is rarely interpreted at once, portions of a program containing multiple lines are often interpreted, and forms can span these lines. However, this said, it does turn out that one of the most common sub-units of a program that students interpret is an individual line of programming. Furthermore, it is here that the interpretation of programs often resembles the interpretation of algebraic expressions. There are lines of programming that are straightforwardly related to algebraic expres-

sions, and these lines are constructed and interpreted in a manner very similar to that of algebra-physics expressions. These observations are not surprising particularly given the fact that my student subjects had significant experience in algebra-physics upon entry in the study.

For illustration, consider Tim and Steve's construction of a simulation of a dropped ball with air resistance. Recall that, using their original simulation of a dropped ball as a starting point, Tim and Steve began by adding a single line to compute the value of a resistance variable:



Tim: The faster you're moving the bigger the – so what is this. So this is – this is something times velocity. . . .

The point here is that this single line of programming was constructed and interpreted in a manner that strongly resembles the construction and interpretation of similar expressions in algebra-physics. As in algebra-physics, I believe that the construction here is based on the DEPENDENCE, PROP+, COEFFICIENT, and IDENTITY FORMS, and on the PHYSICAL CHANGE device.

There is actually a minor difference here between the programming and algebra cases that bears mentioning. Notice that, in the programming case, Tim and Steve wrote a specific numerical value for the coefficient, rather than a symbol like 'C' or 'K.' This turns out to be indicative of a rather consistent difference between programming and algebra-physics. As I will discuss later, it was somewhat more common for the programming students to work with specific values for quantities, rather than with those quantities represented by letters.

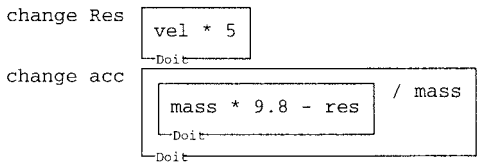
It is important to note that the first argument in a **change** statement must be a single variable; it cannot be an expression. For this reason, the algebraic counterparts of statements involving the change **command** are limited to expressions of the form

$$x = [\ldots],$$

the class of expressions that can be interpreted in terms of the IDENTITY form. Notably absent from this class of expressions are many that would often be interpreted in terms of BALANCING, such as those that state the equality of forces. This is not to say that we could not devise a means of working expressions of this sort into simulation programs. I will say more about this when I compare programming-physics and algebra-physics.

6.2. *The Interpretive Devices of Programming-Physics*

Now I want to discuss how the understanding of programs differs from the understanding of equations. To begin, I return to Tim and Steve's work on the Air Resistance task. After they had produced and tested their final simulation, shown in Figure 7, I asked Tim and Steve to explain why their simulation reaches a terminal velocity. Tim explained as follows, pointing to two statements within the simulation:



Tim: Well it reaches terminal velocity when the resistance is
equal to the weight. And, the weight's constant. . . . Tim: Well, basically, as this – this [points to first line] eventually gets close to this number [points to **mass*9.8**] and the difference [points to **mass*9.8** − **res**] becomes next to nothing. And once that happens there's no more change in velocity. Velocity becomes constant.
. . .
Tim: Well, whatever force vector's going down, that um, cause it to, um, (3.0) accelerate. And, as you - when you accelerate, the resistance gets bigger. So, it kinda like – so, every time you accelerate a little more, the resistance gets a little bigger. So eventually, um, this grows and this grows and eventually it'll reach a point where they're equal.

What Tim is saying here is that, as the velocity increases, the value of the **Res** variable will increase until it equals the value of **mass*9.8**. At that moment the expression in the second line becomes zero and terminal velocity has been reached.

There are two important points to be made about this passage. The first point is that, although it involves multiple lines, it is really quite similar to the PHYSICAL CHANGE interpretations we encountered in algebra-physics. As in those interpretations, Tim looks at a symbolic display and talks through how the variables in that display vary through the time of the motion. A second point to be made about Tim's interpretation is that, if his interpretation is at all representative, then understanding a program may mean focusing on only a small portion of the program.

6.2.1. *A New Narrative Device: Tracing*

I have just taken pains to argue that not all algebra-physics interpretations involve mentally stepping through a program – what has sometimes been called "mental simulation" (e.g., Letovsky et al., 1987). However, there is one representational device that can reasonably be considered to be a variety of mental running. Interpretations based on this new

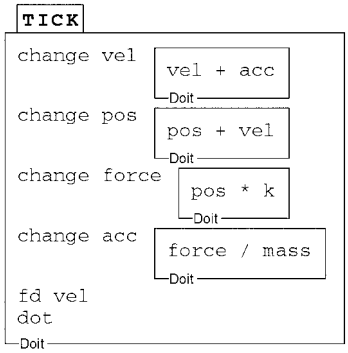


Figure 8. Ned and Dave's tick procedure for the Spring simulation.

Narrative device, which I call TRACING, take their narrative flow from the line-by-line structure of a program.

For a first example of a TRACING interpretation, I turn to an episode involving the work of Adam and Jerry on the Spring simulation. In this task, the students are asked to produce a simulation of a mass that oscillates horizontally on a spring. Early in their work on this task, when they were operating in a somewhat experimental mode, Adam and Jerry produced a simulation that did not make use of the Tick Model structure:

```
repeat 50 [
          change pos 50
          dot
          change pos 0
          dot
          change pos -50
          dot
          ]Doit
```

Their idea, in writing this short program, was to simulate an oscillation by sequentially placing the block at three positions: +50, 0, and then −50. If this was done repeatedly, then the block would appear to oscillate, albeit in a somewhat jumpy and discontinuous manner. After this program was written, Adam checked that he had something sensible by tracing through what they had written:

Adam: Make a dot right there. Change to fifty. Make a dot. [Points to the right of center in the graphics box.] Go back to here make a dot. [Points to center of graphics box.] Go here, make a dot. [Points to left of graphics box.] And repeat.

In this interpretation, Adam steps through the program one line at a time. As he does, he points into the graphics box on the display, moving his hand as the block would move for each statement. This is a TRACING interpretation, a narrative interpretation that follows the timeline structure of the written program.

To further clarify the nature of TRACING, it helps to distinguish it from PHYSICAL CHANGE, since both of these devices involve a description of a phenomenon that follows through the time of the motion. The difference enters in how these descriptions are related to the written expressions. In the case of PHYSICAL CHANGE, one or two lines in the program are selected, and then the interpretation describes how quantities in the expression vary through time. In contrast, rather than focusing on how the quantities in selected lines vary through time, a TRACING interpretation proceeds, line-by-line, through a program. Compare Tim's interpretation of the Air Resistance program described just above. The action in that interpretation concerns how the value of the **res** variable changes through time, with attention focused on two statements. But the action in Adam's TRACING interpretation flows from line to line in the procedure.

Interestingly, stepping through the lines of a program will not, in all cases, correspond to stepping through the time of the motion. For illustration, consider Ned and David's interpretation of the **tick** procedure shown in Figure 8, which has been modified for the Spring simulation:

David: See, cause if we start off with zero acceleration, velocity would be velocity. And it'll go once. Position'll be position plus velocity. Force would – (0.5) position times K. And then acceleration'd be force divided by mass. And then, it'll use that acc, right, the next time around?

In this passage, David proceeds through the lines in **tick** under the assumption that the acceleration is zero. The point about this interpretation is that, although it follows the lines in the program, it is not precisely correct to think about the sequence of operations as happening *through time*. In the physical system that this simulation purports to model, all of these parameters are constantly changing; they do not change one at a time as in this program. In essence, the act of programming requires the students to explode each instant of the motion into a series of actions that happen through what I will refer to as "pseudo-time" (deKleer and Brown, 1984).

6.3. *The Symbolic Forms of Programming-Physics*

In this section, I will look at how the symbolic forms of programming-physics differ from those of algebra-physics. For the most part, the new forms in programming-physics involve a sensitivity to line-spanning structuring in programming. First I will describe a new cluster of forms, the Process Cluster. Then I will describe one additional form that did not appear in algebra-physics, SERIAL DEPENDENCE, which is an important line-spanning addition to the Dependence cluster.

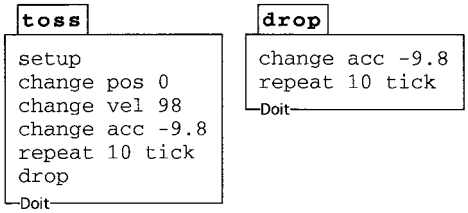


Figure 9. Tim and Steve's first working Toss simulation.

6.3.1. *A New Cluster: The 'Process Cluster'*

When programs are being constructed, programming *requires* that the statements be arranged in a sequence. For the most part, I will account for the ability of students to produce novel arrangements of statements by positing a new cluster of forms that I call the 'Process Cluster.' To introduce the forms in this new cluster, I want to discuss how Tim and Steve adapted their simulation of a dropped ball for Task #3, which asks them to simulate a motion in which a ball is tossed straight upward and then falls back down. Tim and Steve's first working version of this new simulation appeared as shown in Figure 9.

Execution of this program starts in the box named **toss**. **Toss** begins by executing **setup** to position the ball at its starting point and orient it facing upward. Then each of the three variables are initialized and **tick** is repeated 10 times, taking the ball up to the apex of its motion. Following the **repeat** statement, there is a single line that calls a procedure named **drop**, which takes the ball from its apex back down to the bottom of its motion.

One thing that needs explaining here is how Tim and Steve knew to order the lines in **toss**. To begin, notice that the first five lines of this procedure, from **setup** to the **repeat** statement, are in line with the initialize-and-then-repeat structure, which we encountered above. Just as this structure is familiar to readers, it was also familiar to Tim and Steve; they had already seen several examples and created a few programs following this structure. Thus, I believe that it is reasonable to hypothesize that the sequence of statements

setup
initialize variables
repeat ## tick

is a structure that Tim and Steve essentially remember; it is part of a known repertoire of program structures. (I will have more to say about this structure in a moment.)

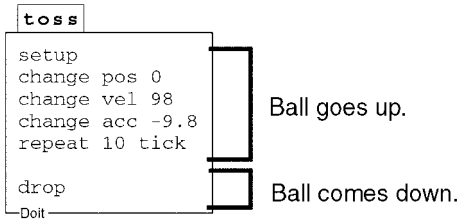


Figure 10. Tim and Steve's Toss procedure broken into two parts.

But in appending a call to **drop** at the bottom of the **toss** procedure, Tim and Steve are departing from the initialize-and-then-repeat structure. How did Tim and Steve know to write this particular line in this location? To an experienced programmer, this may seem like an almost trivial move, hardly worthy of an explanation. But, I hope to show that working to describe the knowledge involved here actually extracts some of the most important differences between algebra-physics and programming-physics.

The key to the explanation here is to note that, in writing the program in this manner, Tim and Steve have broken the motion into two parts, the upward motion to the apex and the downward motion. With this in mind, the **toss** procedure can be understood as consisting of two sequenced parts that correspond to these two portions of the motion. This is illustrated in Figure 10.

When a procedure is seen in this manner, as consisting of two or more sequential pieces that correspond to segments of a motion, I say that the SEQUENTIAL PROCESSES form is active. The entities in the conceptual schema associated with SEQUENTIAL PROCESSES are two or more *processes*, and the relationship that holds among these processes is simply that they are *ordered*. Furthermore, the associated symbol pattern is:

[. . .]
SEQUENTIAL PROCESSES [. . .]
[. . .]

In this notation, "[. . .]" stands for a single programming statement or a group of statements.

Notice that, except for ordering, SEQUENTIAL PROCESSES treats all of the processes involved as equivalent. As far as the schematization associated with this form is concerned, none of the processes plays a special role. In contrast, we can imagine that it would be possible to develop standard ways of breaking a motion into sub-processes in which the sub-processes are non-equivalent.

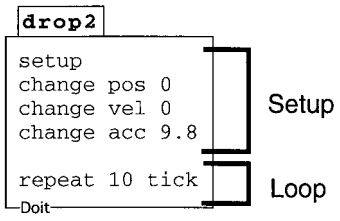


Figure 11. The setup-loop structure in Tim and Steve's drop simulation.

One such breakdown, associated with the initialize-repeat structure, is what I call the SETUP-LOOP form. In SETUP-LOOP, the program is seen as decomposed into a portion that gets everything ready for the motion – for example, positioning the sprite and initializing variables – followed by a **repeat** statement that repeatedly executes a procedure that specifies what transpires at each moment of the motion. To take a simple example, Tim and Steve's drop simulation can be seen in terms of **setup-loop** (refer to Figure 11). Note that, since SETUP-LOOP breaks down a motion into sub-processes associated with program parts, it may be thought of as a special case of SEQUENTIAL PROCESSES.

Before concluding this section, I want to briefly connect to some closely related work. Other researchers have argued that programming knowledge includes what have been called "templates" (Linn et al., 1987; Mann, 1991) or "plans" (Ehrlich and Soloway, 1984; Soloway, 1986). These templates are schematic structures – like the ones I have been describing – that constrain, at certain levels, the programs that experts and students write. In the template literature, it is usually emphasized that templates can exist at many levels; they can exist at the level of individual programming statements, or closer to the level of entire programs. My position here is that, at least to a first approximation, it is reasonable to view the Process forms – and perhaps programming forms, in general – as a particular sub-category of template. The templates in this sub-category specify programs at a certain level, and have a certain degree of specificity; they have only a small number of components, each of which is quite generic.

There is another way to understand the special characteristics of this sub-category of template. In the templates literature, it is often emphasized that templates are associated with *goals* or *functions*. In my account of forms, the story of use is somewhat different. When constructing a program, a form is activated because of the meaning of the construct – the associated conceptual schema. Thus, Process forms are a particular sub-category of template that correspond to a conceptualization of a process, and thus allow the generative construction of simulations out of meaningful components.

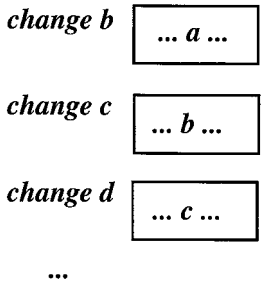


Figure 12. The serial dependence symbol pattern.

6.3.2. *A New Dependence Form: Serial Dependence*

In programming-physics, the notion of dependence gets extended in an interesting way. When writing simulations, students were often concerned with *chains* of dependent quantities, such as **d** depends on **c**, which depends on **b**, which depends on **a**. Within a program, these chains show up as the symbol pattern shown in Figure 12. In this programming symbol pattern, **b** is computed from **a**, **c** is computed from **b**, **d** is computed from **c**, etc. This sequencing of linked dependence relations, which students wrote into and read out of programs, is the meaningful pattern I call SERIAL DEPENDENCE.

We already encountered an episode involving SERIAL DEPENDENCE above, when I gave an extended description of Tim and Steve's work on the Air Resistance simulation. Recall that, in that episode, Tim raised the issue of how the lines in their modified **tick** procedure should be ordered. With their **tick** procedure as shown in Figure 6, Tim commented:

> Tim: No we have to (then account) in the acceleration. So we have to change the acceleration. Do we change the acceleration before we change the velocity? [Pointing into **tick** box here.] Or do we change acceleration after we change velocity? Which also changes the resistance. Oh, I'm getting confused. What's the order in which you change things?
>
> . . .
>
> Tim: Well you have to change resistance because you change the velocity, because (0.5) the velocity (1.0) – the new velocity is dependent upon the acceleration which is dependent upon the resistance. So we should change resistance, then acceleration, then velocity. Right? Does that make sense, or am I just talking nonsense.

The question that Tim and Steve were wrestling with was, given that a number of quantities must be computed in the **tick** procedure, "What's the order in which you change things?" In answering this question, they decided to compute **res** (a quantity associated with the air resistance) from the current value of the velocity. Then they got an acceleration from this value of **res**, and finally a new velocity and position from the acceleration.

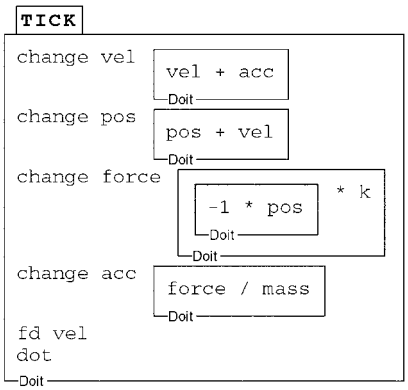


Figure 13. Ned and David's **tick** procedure for the Spring simulation.

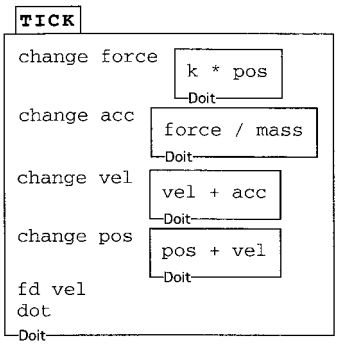


Figure 14. Adam and Jerry's tick procedure for the Spring simulation.

Tim's assertion is that this ordering somehow actually reflects the structure of *dependence* in this situation, "the new velocity is dependent upon the acceleration which is dependent upon the resistance." Of course, he does not provide much of an argument for this view in the above passage. Nevertheless, this ordering does turn out to be in agreement with one important way of conceptualizing the structure of Newtonian physics: Find the force using a 'force law' and then use $F = ma$ to find the acceleration. I will have more to say about this below.

SERIAL DEPENDENCE issues arose frequently during student work on the Air Resistance simulation, as in Tim and Steve's discussion above, and also in the context of the Spring simulation. Ned and David's brief discussion below is another example. In this example, Ned and David decide on a particular dependence structure: the position determines the force which determines the acceleration. (Refer to Figure 13.)

Ned: Does the force change because the acceleration changes?
David: The force changes 'cause X changes. And so that, and so =
Ned: = That causes acceleration to change.

Similarly, the question of which quantities should determine which others played a central role in Adam and Jerry's work on the Spring simulation. Throughout their work, Adam strongly argued that the position was at the root of the dependence structure:

Adam: Well basically, the position is, um, the acceleration, everything, is based on where this is, the position.
. . .
Adam: And the position is the most important thing here. If we made it the other way around, we don't care about the acceleration, we care about where it is. See acceleration derives from the position.
. . .
Adam: That's the best way. It's easier just to think about it as position affecting acceleration. Than acceleration affecting position.

In accordance with this viewpoint, Adam and Jerry settled on the version of the **tick** procedure shown in Figure 14. In this and the preceding passages, the students were engaged with a quite unusual set of issues. The question under consideration – which quantities determine which others – is not a question that arises very often in algebra-physics. In fact, such a question may not even make sense within algebra-physics.

7. COMPARISON OF PROGRAMMIMG-PHYSICS AND ALGEBRA-PHYSICS

My discussion has now reached its apex. I have presented an analysis of both algebra-physics and programming-physics in terms of interpretive devices and symbolic forms. In this section, I will now draw on those analyses in order to compare the two practices. I will present, first, a comparison based on interpretive devices, followed by a comparison based on symbolic forms.

7.1. *Comparison Based in Interpretive Devices*

Recall that I began by asking whether programs would be easier to understand than equations. With the introduction of my framework, this informal question can now be refined. We can ask: How does interpretation differ in programming-physics versus algebra-physics? Is there any sense in which the interpretive devices in one practice are easier to execute or easier to learn? We have already seen that the comparisons implied in these questions are not going to be simple and straightforward. It is not the case that

only programs are interpreted; we saw that equations are also interpreted by students, and often in a manner that is similar to the way that programs are interpreted. Nonetheless, I believe that there are still some important differences to be drawn out.

7.1.1. *A Shift to Time-Varying or 'Physical' Devices*

In my initial hypotheses, I postulated that programs would be easier to understand because they can be mentally run. In my more formal analysis, the mental running of programs was captured by a particular interpretive device, TRACING, that only shows up in programming-physics. The existence of this device that is particular to algebra-physics constitutes one obvious difference between programming-physics and algebra-physics. However, the importance of this difference is not immediately clear. TRACING is only one of a number of narrative devices, and all of the other narrative devices show up in algebra-physics. Thus, the existence of one additional narrative device in programming-physics might not be very important.

To further explore this issue, recall how the various devices in the Narrative cluster differ. In PHYSICAL CHANGE, the narrated changes follow the motion under study. For example, if we are examining a ball falling under the influence of air resistance, then we might narrate how the velocity and the air resistance force change as the ball drops. In contrast, a CHANGING PARAMETERS interpretation describes changes that could be understood to be occurring across instances of a motion. For example, again considering the case of a ball dropped with air resistance, a CHANGING PARAMETERS interpretation might describe how the motion would differ if the ball was heavier. Given these descriptions of the three Narrative devices of algebra-physics, we can see that there is a sense in which TRACING and PHYSICAL CHANGE go together: both of these devices involve changes through the time (or pseudo-time) of the motion under study.

With this in mind, I want to introduce an observation drawn from the systematic analysis in Sherin (1996). It turns out that TRACING and PHYSICAL CHANGE interpretations are both common in programming-physics and that, together, they constitute a large percentage of all interpretive utterances. In contrast, PHYSICAL CHANGE interpretations are relatively uncommon in algebra-physics (and, of course, there are no TRACING interpretations). Thus, what we are seeing here is a general shift toward what I call 'physical' narratives – narratives that follow the motion under study. This is an important observation, and it suggests a general difference in how programming expressions tend to be understood. As we will see, this

is not too surprising given an overall learning toward static situations in algebra-physics.

7.1.2. *Programming and Embedding in Particulars*

In this sub-section, I want to draw out a difference that has not been well-captured by previous parts of my discussion. Note that, in algebra-physics, it is possible to use equations to solve problems in which the physical situation is only weakly specified. For example, in the Mass on a Spring task that I gave to my algebra subjects, the students were asked to determine how the amount of stretch depends on the mass of the object and the spring constant. However, they were not given any specific values for these parameters. Instead, they simply derived the equation $x = {}^{mg}/_{k}$ and then discussed the properties of this expression. In this way, they were able to answer the question without specifying the numerical particulars of the physical situation.

Furthermore, you can use algebraic expressions in a manner that is farther removed from even sketchily described physical circumstances. For example, it is possible to use Maxwell's equations (a set of very general relations that describe phenomena relating to electricity and magnetism) to derive the fact that waves can propagate in electric and magnetic fields. In that case, we are deriving a very general feature of electric and magnetic phenomena; we do not need to be talking about any particular physical circumstance.

In contrast, programming-physics – at least the practice used in this study – requires one to embed statements in particulars. For example, when creating their Spring simulations, the students in my study were forced to invent numerical values for all of the parameters that appeared in their programs. If they did not invent these numerical values, then they could not run their programs. In fact, partly for this reason, I saw that Special Case interpretations were more common in programming (Sherin, 1996). Because programming already requires you to deal with equations in a particular numerical regime, Special Case interpretations may be more natural.

There is still more to this beyond the fact that programming requires you to pick specific values. When you write a statement in programming, that statement must be embedded within a program, and it must be embedded in a specific manner. You cannot have the programming statement **change acc force/mass** written just anywhere on the computer display, it must be embedded within a program, doing a particular job. The point here is that the existence of these particulars in programs might make interpretation easier. For example, because individual statements are

embedded within programs, this may help to suggest that the statement be interpreted in a particular manner. I will say more about this below.

7.1.3. *Are Programs Easier to Interpret than Equations?*
Are programs really easier to interpret than equations? Given the observations I have presented, there are reasons to conclude that programs are easier to interpret, and reasons to conclude that they are trickier to interpret. To begin, we must keep in mind that the interpretation of programs was not always very different than the interpretation of algebraic expressions. In many cases, individual programming statements and collections of statements were interpreted much like their algebraic counterparts, with the rest of the program fading into the background. This observation seems to suggest that programs are no easier to interpret, and maybe even a little harder, since the lines to interpret in a program must first be selected.

However, on the other side of this debate is the observation that the backdrop provided by the rest of a program and the programming context may help by providing an orientation to the lines that are focused on. This is essentially the point I made just above. The fact that programming statements are always embedded within a program in a particular manner provides a leg-up on interpreting those statements. Less has to be done to invent a stance, at least once the program is written, since the interpretation is already partly suggested by the role that the statement plays within the program.

This observation is actually a corollary of a more fundamental point. In general, interpreting any specific symbolic expression will be easier when there is more support lying around in the context, and a person thus has to do less to invent an interpretive framing. Furthermore, this is true not only of individual expressions, but also of classes of expressions: Any class of symbolic expressions will be easier to interpret to the extent that it is embedded in practices that provide natural modes of interpretation. Yet another way to say this is to state that, to some extent, the ease of interpretation of symbolic expressions depends on the existence of naturally available and useful interpretive devices. And there actually is some evidence that programming has an edge in this regard. The fact that the interpretation of a programming statement can be framed by the running program is an example of this sort of natural support. Because of the fact that programs can be run, there is a naturally available representational device – the TRACING device.

In conclusion, there are points on both sides of this debate. In some respects the interpretation of programs is not altogether different than the

interpretation of equations, and some cleverness is required in selecting lines and pushing other lines into the background. However, because programming statements are embedded in particulars there is more support for interpretation. Furthermore, there are some naturally available and very powerful interpretive devices that simulation programmers can fall back on in interpreting programs.

7.2. *Comparison Based in Symbolic Forms*

7.2.1. *Unbalanced* BALANCING

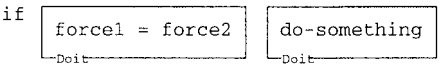
My comparison of programming-physics and algebra-physics now turns to the symbolic forms associated with each practice. It is profitable to begin this discussion with a consideration of the BALANCING form. In contrast to a relative prevalence in algebra-physics, I never encountered the BALANCING form in my analysis of the programming corpus. In part, this must have to do with the specific nature of the tasks I selected, and with some details of the particular practice of programming-physics I designed for this study.

However, I believe that this observation also points to some real fundamental differences between the two practices and their associated symbolic languages. In particular, situations involving BALANCING turn out to be somewhat easier to deal with in algebra-physics than situations that do not involve BALANCING. For example, when we have balanced forces, then we can just set the two forces equal and solve, in the process avoiding differential equations. Because of the simplification that results, the practice of algebra-physics tends to seek out situations that can be understood in terms of BALANCING. In contrast, BALANCING situations are not especially easy or natural in programming-physics. Thus, we see a possible strong effect, rooted in the representational languages, in the type of meaningful structures we seek out in the physical world.

I do not mean to imply that the use of algebra constrains physics to the study of unimportant phenomena. In fact, there is reason for us to have special interest in static phenomena – cases in which all objects in a system are at rest relative to each other. For example, these cases happen to be very important to engineers and architects, who need to design buildings and bridges that do not move (i.e., that do not fall over).

The situation in programming is quite different. To begin, notice that there is no way to write symmetric BALANCING relations like $F_1 = F_2$, at least within the practice of programming-physics that I employed in my study. In particular, the programming statement **change F1 F2** is not equivalent to $F_1 = F_2$. Rather than stating a symmetric constraint, this

programming statement specifies a directed action: Assign the value in **F2** to the variable **F1**. We could attempt to modify our practices so that such differences in tendencies were moderated. For example, we could expand our practice of programming-physics to include 'conditional' statements, such as:



This statement says to execute the procedure named **do-something** if the variable **force1** happens to equal the variable **force2**. If we allowed these conditional statements in our practice of programming, there would be a niche for symmetric equality relations that might lend themselves to BALANCING interpretations.

However, I believe that even in modified practices of this sort, BALANCING will still play a less prominent role than it does in algebra-physics. This is true, if for no other reason, because we *can* deal with time-varying situations so easily. There is simply no reason to lean as heavily on cases in which influences happen to balance.

7.2.2. *Ordering and Processes*

In my original informal hypotheses, I conjectured that there would be important differences between programming and algebra traceable to the fact that the lines in programs are strongly *ordered*. In part, the effects of this strong ordering are reflected in the existence of the Process Cluster, a whole new cluster of forms that are unique to programming-physics. The existence of this new cluster is extremely important; as I will argue, it suggests that a whole new wing of intuitive knowledge is picked up and strongly supported by programming.

The points here are so important that I want to support them with a careful discussion of an example episode. This episode involves Tim and Steve's work on the Toss simulation, which we briefly encountered above. Here I want to go through this episode in some detail, telling the whole story so that I can extract the relevant morals.

Upon starting work on the Toss, Tim and Steve commented that the second portion of the motion, during which the ball drops from its apex to the ground, would be the same as the motion produced by their Drop simulation, which they had just completed. This led Tim and Steve to copy their work from the Drop simulation and past it into their current working environment. Then they worked to fill out the **toss** procedure and adapt **drop** to be appropriate for its current use. The result was the program shown in Figure 15. Recall that, above, I argued that the **sequen-**

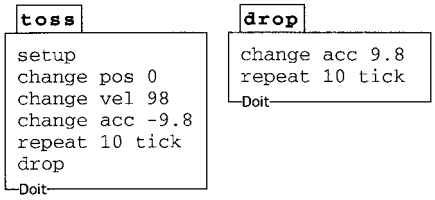


Figure 15. First version of Toss simulation.

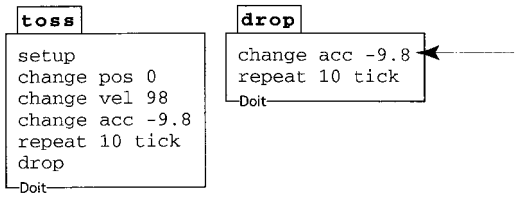


Figure 16. Toss simulation with modified acceleration line.

tial processes form was involved in Tim and Steve's construction of this **toss** procedure. The story I told there was that these students saw the simulation as broken into two parts, a part that takes the ball up and then a call to **drop**, which is supposed to take the ball down (refer to Figure 10).

When Tim and Steve executed this simulation, the sprite began by moving upward in the graphics box, slowing down along the way. This part was as expected. But after the sprite stopped at its apex, it began to move upward again, accelerating off the top of the graphics box. Looking at the graphical display, it was obvious to the students that something was amiss:

Steve: Oh, shoot.
Tim: What happened?
Steve: It kept going the same way.

The reason for this incorrect behavior is the change in the sign of **acc** at the apex of the motion. To explain: The ball/sprite starts off at the bottom of the graphics box, pointing upward. This means that the downward direction is essentially defined to be negative. Thus, when the program starts, the ball's initially positive velocity causes it to move upward, and the initially negative acceleration decreases this velocity, making the velocity smaller and smaller until it is zero at the apex. At this point, for the ball to come down, we would need the velocity to become negative so that the ball moves in the negative direction. This requires that the acceleration continue to be negative. If the acceleration is changed to have a positive value at this point, as in Tim and Steve's simulation, this makes the velocity become positive again, and the ball continues upward.

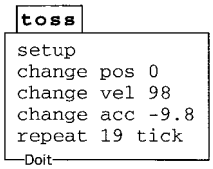


Figure 17. Final version of the toss simulation.

Without making any comments whatsoever, Tim and Steve immediately modified their **drop** procedure so that it made the acceleration -9.8, instead of 9.8. Thus, their program looked as shown in Figure 16. This program, when run, produced behavior that was essentially correct and Tim and Steve were largely satisfied. However, notice that, because the **change** statement in **drop** now changes the acceleration variable to -9.8 – the value it already contains – this statement has no effect and can be safely deleted. Furthermore, when this statement is eliminated, the above program essentially contains two sequential **repeat** statements, both of which repeatedly execute the same **tick** procedure. These two statements can thus be combined into a single longer **repeat** statement. Tim and Steve recognized these points soon after completing their successful simulation:

Steve: So we don't even need to have that change, huh? [Points to **change acc** line in **drop**.]
Tim: That can't be right.
Steve: Cause it's still the same.
Tim: Then why does it go back down?
Steve: I don't know.
Tim: Oh, cause we have an initial velocity right. It starts out positive, it just keeps getting more and more negative.
Steve: Oh. So you could just repeat that nineteen times. [Points to **repeat line** in **toss**.]
Tim: Yeah.

After this exchange, Tim and Steve acted on these observations to produce the final version of the simulation, shown in Figure 17, which generates identical behavior to their previous simulation. After confirming that this new version of the simulation worked, Tim commented:

Tim: We don't need this. [Points to **drop** procedure.] That's right, cause gravity's constant.

Now I want to take some time to reflect on what happened in the above episode. Initially, Tim and Steve had a program that broke the motion into two parts, a part that takes the ball up to the apex and a part that takes the ball back down. Then this program was transformed into a new version that treated the motion as a single whole. These two views of a toss are really quite dramatically different, but is there a sense in which either version is preferable? Have Tim and Steve learned anything important here?

To begin to answer these questions, let us first look at these issues from the point of view of a physicist. Note that, throughout the motion, the only force acting on the ball is the force of gravity, $F_g = mg$. This force can be used to compute the acceleration of the ball at any time using the relation $F = ma$. Thus, there is a deep sense in which the motion is the same at every point, and it is not 'physically natural' to break the motion into two parts. This does not mean that it is incorrect to talk about the motion as having two parts; there may very well be occasions in which this point of view is helpful and illuminating. However, if a student does learn to see the toss motion as a single whole, then they may be on the way to learning something profound. The power and subtlety of the formal physics approach here is that it reveals a common, underlying mechanism that generates the entire motion.

Now let's think about where Tim and Steve stand with respect to these issues. Initially, it was not obvious to them that the toss motion could be treated as a single uniform process. This may seem a bit surprising, given the fact that Tim and Steve have almost certainly received direct and fairly extensive instruction relating to this issue. That Tim and Steve have received such instruction is somewhat confirmed by Tim's final statement: "That's right, cause gravity's constant." Nonetheless, in their first pass at this simulation, Tim and Steve thought that they needed to alter the sign of the acceleration at the apex. Did they just forget or was there still some lingering sense in which they didn't really 'get it'? Referring to the research literature, there is substantial corroborating evidence that many physics students cannot give accounts of a toss in which the upward and downward motions are treated equivalently. This asymmetry is apparent, for example, in both McCloskey's (1984) and di Sessa's (1993) accounts of student's explanations of tossed objects.

The point here is that programming forces a student to engage with these issues in a way that algebra-physics does not. The question of how and whether to slice up the motion has clear and direct implications for the program; you have to decide whether to have one loop or two. In algebra-physics, as it is usually practiced, the relevant differences are not so clearly manifested as differences in the external representation. Given this observation, it is not too surprising that algebra-physics instruction leaves some undiscovered intuitive cobwebs in this territory, and that the act of programming tends to expose these cobwebs.

Tim and Steve's work on the Toss simulation was not at all anomalous. In fact, three of the four other pairs explicitly discussed the issue of whether the simulation should involve one or two loops. Furthermore, nearly the same issue, in one form or another, arose for all pairs while

working on the Spring simulation. As for the Toss simulation, a single **tick** procedure and repeat loop can be used to generate the entirety of the Spring simulation. But this was far from obvious for the students in my study; pairs frequently discussed various ways of breaking an oscillation into parts. Then, as their work proceeded, they progressed to a solution in which the entire motion was generated by a single repeat loop.

7.2.3. *Directedness and Causation*

Another element of my original hypotheses was that the directed aspect of programming statements would lead to a greater role for causal intuitions in programming. Since there is no symbolic form that I labeled the 'cause' form, the connection of causal intuitions to my analysis is not immediately obvious. Among the forms that I identified, the ones that I believe are most closely related to causal relations are the Dependence forms. A little explanation is required here to make this relationship between dependence and causation clear.

First note that causal relations are asymmetric in the sense that when I say that *X causes Y*, this is not equivalent to saying that *Y causes X*. Furthermore, there is a family of relations that are asymmetric in this sense. Dependence is perhaps the weakest of these relations; when we say that one quantity depends on another, all we are saying is that changing the value of the second quantity will alter the value of the first quantity, all other things being equal. A slightly stronger asymmetric relation between quantities might be called 'determining'. If quantity *X determines* quantity **Y**, this means that knowing the value of *X* (perhaps together with some other quantities that we know about) is enough to uniquely specify the value of *Y*. Finally, following Horwich (1987), *causal* relations can be understood to be like determining relations, but with an extra "ingredient" added, such as a requirement of time-ordering.

The point of the above story is not to provide a rational accounting of dependence and causation. Instead, I simply want to argue that observations pertaining to the Dependence forms are relevant to what I informally called 'causal intuitions' in my original conjectures. More specifically, I want to assume that 'dependence' can be taken as a name for a weaker and more encompassing category of relations that includes causal relations. So, at least tentatively, we can rephrase the questions here as follows: Are there differences between programming-physics and algebra-physics relating to Dependence forms? Did these forms appear more often in programming or was the sense of dependence somehow stronger?

We should begin by noting the simple fact that, in the case of algebra-physics, students did make interpretations that treated the two sides of an

expression asymmetrically. This is an extremely important observation. Although algebraic equations are always, in a certain sense, symmetric, it does not seem to be overly difficult for students – at least the relatively advanced students in this study – to read asymmetries into these statements. For example, although the equal sign in $F = -kx$ specifies a symmetric relation, it is not especially difficult for students to read this asymmetrically as 'force is dependent on position.'

This suggests that the mere fact that programming statements are directed may not be as consequential as I initially conjectured. Nonetheless, I still want to argue that there is, in fact, an important sense in which programming-physics draws more strongly on causal intuitions. In my discussion of SERIAL DEPENDENCE, we saw that programming forces students to engage with a unique set of issues-issues pertaining to which quantities change first and which quantities determine which other quantities. For example, in their work on the Spring simulation, we saw that students discussed whether it made more sense for position to determine acceleration, or acceleration to determine position. The importance of this observation is driven home if we think about algebra-physics. When working within algebra-physics, we typically adopt the stance that many quantities are changing constantly and simultaneously. For example, when we write $F = ma$, we do not need to think about whether force or acceleration changes first. In fact, since these and other quantities are changing constantly, this question does not even make sense within algebra-physics. In contrast, programming forces you to take changes that happen simultaneously and break them up into ordered operations. And, in doing so, you have to decide which operation happens first. This is a step toward an ordered, causal world.

7.2.4. *Are Causal Intuitions Problematic for Physics Learning?*

At the start of this paper, I said that I would not only ask how programming-physics differs from algebra-physics. In addition, I want to know whether programming-physics is a *respectable* form of physics. Suppose we accept that programming tends to draw on causal intuitions in a certain manner, and that it forces programmers to break up simultaneous changes into sequential operations. Then we have to ask: Are these properties of programming desirable from the point of view of physics instruction?

One reason we might be prompted to ask this question is that there are actually reasons to believe that the use of causal intuitions in physics may be dangerous or even incorrect. For example, an algebra-physics traditionalist might argue that it is incorrect to say that forces *cause* accelerations.

The equation $F = ma$ is symmetric in this regard; it doesn't say that forces cause accelerations any more than it says that accelerations cause forces. Thus, in introducing causation or other asymmetric relations here, we may be adding information that is not really part of a 'correct' physical description of the world.

There are a number of responses to this objection. First, we must keep in mind that a concern with how someone *understands* physics is not the same thing as a concern with what is *correct* physics. Even the best physicists may sometimes think of forces as causing accelerations, even if they somehow know that this is not strictly correct. If it is the case that physicists think causally, then it may not be a problem to teach students to think causally.

Furthermore, I believe that it is immensely plausible that physicists draw on causal and related intuitions. One reason that I believe that it is plausible is that there are many important uses for causal intuitions. For example, notice that, although it may not be right to say that forces *cause* accelerations, it is not true that there is no asymmetry in the relation between the concepts of force and acceleration. At the least, there is an ontological asymmetry – forces and acceleration are two different sorts of entities.

In addition, students do not always keep this ontological asymmetry straight. One way that this problem manifests itself is in misinterpretations of the equation $F = ma$. Sometimes students look at $F = ma$ and see it as a statement that two forces are in balance (Sherin, 1996, in press; di Sessa, 1984). This is incorrect because only the F in this equation is directly associated with a force or forces that exist in the world. Thus, there is an asymmetry here that causal intuitions could help students to keep straight: If you have in mind that forces cause accelerations, then you are less likely to misinterpret this equation. This is the sort of place that causal/dependence intuitions are useful. Instead of writing $F = ma$ and seeing it in terms of BALANCING, it is likely better for students to write $a = {}^{F}/_{m}$ and recognize the DEPENDENCE form, with acceleration dependent on force. (See di Sessa, 1993, for a similar argument.)

7.3. *Some Differences not Captured by the Analysis: Channels of Feedback*

There are differences between programming-physics and algebra-physics that are not captured by my main analysis. For example, I could have paid more attention to what each of these notational systems *allows*. Algebraic notation allows you to manipulate expressions to derive new expressions, and programs can be run to generate dynamic displays. These differences

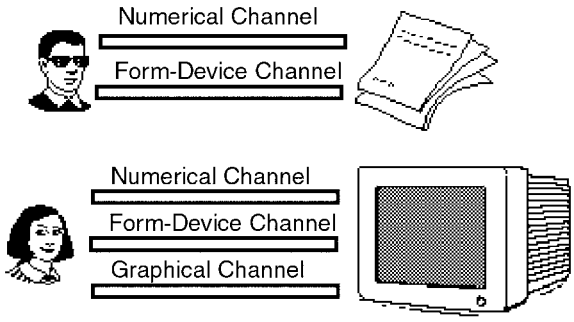


Figure 18. Channels of feedback in programming and algebra.

in what is allowed are extremely important for several reasons. First, they in part dictate what you can *do* with each of the notational systems. But, even more important from our point of view, they will also have effects on students' understanding. This sort of second-order effect on understanding is not directly captured by my form-device analysis.

To get at these additional issues, I am going to take the theoretical work that I have already done and I will embed it in a somewhat broader view. At the heart of this broader view is a notion that I call 'channels of feedback'. To explain channels of feedback I want to begin with an image of two students (refer to Figure 18). One of these students is looking at a sheet of equations that he generated on the way to solving a problem. The other student is looking at a simulation program that she just wrote. Now I want to ask the same question about both of these two students: How will each of them decide if what they have just done is correct or 'makes sense'?

Both of these students have a number of possible approaches they can take in deciding if they are satisfied with what they have done, and each of these alternative approaches involves a different 'channel of feedback'. For example, one way that the algebra student can draw conclusions concerning the sensibility of expressions involves what I call the 'numerical' channel; he can plug in numbers and then judge the reasonableness of the numerical result, a procedure that essentially gives indirect evidence as to whether the expression itself is also reasonable. The thing to notice about the numerical channel is that it allows judgments to be made concerning the sensibility of an expression without any attention being paid to the actual structure the expression.

Of course, there are other ways for algebra students to judge the sensibility of their expressions. In fact, a large part of this work has been to show that students do have the ability to judge expressions by looking directly at their structure. I call this channel – which was the focus of the majority of this paper – the 'form-device' channel.

These same two channels – numerical and form-device – are also available to the programming student in Figure 18. The preceding sections can be taken as an argument that forms and devices play a role in how programs are understood. In addition, the students in my study did often look at the values in their variables in order to decide if their simulations were correct.

Now, the central observation here is that programming has an important channel of feedback that is not available in algebra-physics, what I call the 'graphical' channel. The point here is that programs can be run to generate dynamic feedback in the form of a moving graphical display. This constitutes a channel of feedback because, when a program is run, a student can look at the display and judge if the simulated motion appears reasonable.

Programming's graphical channel has a number of very nice properties. One of these properties is that, in certain respects, looking at the dynamic graphical display is like looking at a real motion in the physical world. This means that, to judge whether these displays are correct, a student can draw on some of the same capabilities that they use in recognizing and judging the plausibility of physical phenomena. Compare this to the form-device channel, which requires a substantial amount of specialized learning.

Finally, I just want to add one other piece to this image of channels of feedback. We can think of algebra and programming as each having some *generative machinery*. In algebra-physics you can manipulate existing expressions to generate new expressions, and you can substitute numbers to obtain results. In programming-physics you can run programs to generate a simulated motion in a dynamic display. So, one way to think of the broader image I am building here is to imagine a core of generative machinery with feedback at different places.

8. SUMMARY AND DISCUSSION

In this paper, I have taken seriously the notion that a programming language can serve as a bona fide representational language for physics. Adopting a tight focus around the representational languages, I considered some of the implications of replacing algebraic notation with a programming language. I began with two informal hypotheses. The first was that programs are easier to interpret than equations, and the second that programming privileges a different intuitive vocabulary than algebra. Furthermore, in order to refine and address these two informal hypotheses, I introduced two theoretical constructs, interpretive devices and symbolic forms.

With regard to the first hypothesis, my conclusions were equivocal. On the one hand, it seems that ascribing meaning to algebraic expressions is

not as difficult as one might initially have thought. On the other hand, the surrounding context provided by a program may provide additional support for interpreting parts of a program. Furthermore, there is some suggestion that interpretations of programs may differ in kind; in the practice of programming-physics studied, narrative interpretations that talked through the time of a motion appeared to be more natural.

With regard to the issue of the intuitive vocabulary privileged by programming, I believe that the analysis has uncovered some very important differences. Programming seems to make close connections to a whole new wing of intuitive accounts of phenomena, based around a process vocabulary. Furthermore, it may build more strongly on causal intuitions, and less on notions in the vicinity of BALANCING.

Given these results, I believe it is not too bad a summary to describe algebra-physics as *a physics of balance and equilibrium*, and programming-physics as *a physics of processes and causation*. Algebra-physics trains students to seek out equilibria in the world. Programming encourages students to look for time-varying phenomena, and supports certain types of causal explanations, as well as the segmenting of the world into processes.

It is important to keep in mind that these are differences in degree only. Within algebra-physics, for example, it is certainly possible for students to talk about processes, even though these accounts may not be as directly reflected in representational forms. Nonetheless, these are still important and fundamental differences – these high-level generalizations reflect real differences in the physical intuition that will result from learning physics through the practice of algebra-physics, or through the practice of programming-physics.

8.1. *Toward a New Type of Whorfian Analysis*

In the remaining two sub-sections of this paper, I conclude with some discussion of the nature of the analysis and claims presented here. I begin with the observation that, in some respects, my conclusions might be seen to have a Whorfian flavor. Indeed, I am arguing that, in a deep sense, the nature of an individual's physics knowledge depends on particular details of the representational language that is employed.

However, I hope that it is also clear that my claims are not as sweeping as many assertions that have been made concerning the relationship between language and thought, or the relationship between literacy and thought. For example, it has been argued that, at least historically, the development of writing led to dramatic intellectual achievements and across-the-board changes in the way that individuals reason (Goody and

Watt, 1968; Olson, 1994). In contrast, I have not made any sweeping generalizations about the nature of all of an individual's knowledge. Rather, the shifts in reasoning patterns I described are only posited for a narrow range of cognitive activities, namely, for the way that individuals understand a class of physical phenomena.

I believe that this is the right way to look for relationships between external representations and thought. There are certainly pervasive and widespread inter-connections of external representations and human knowledge. Much of our knowledge simply must be adapted for thinking and behaving in a world with representations. However, even if we believe that external representations thoroughly infiltrate thought at a fundamental level, there is no need to suppose that any particular external representations, including spoken language, are uniquely responsible for defining human thought, or that they have effects on the broad character of human thought.

Instead, I think that there are relationships between human knowledge and external representations at a fundamental level, but that these relationships are local and multifarious. Thus, to explore these relationships, we will simply need many more programs of the sort performed here. However, there is the possibility of some great rewards if we undertake this sort of program across a number of domains. If we look across multiple projects of this sort, it may be possible to say something about typical ways in which external representations infiltrate thought – typical mechanisms and patterns of interaction – even if we cannot make sweeping claims about how the broad character of human reasoning is tied to any individual representations.

8.2. *Final Discussion: What Sort of Analysis Was This?*

What sort of analysis was this? I have self-consciously not adopted a 'transfer' perspective, one that focuses on whether teaching programming-physics would help students to more easily learn traditional algebra-physics. Nor have I undertaken to produce a comparative analysis based on any sort of outcome measures. In other words, I did not develop a single metric for evaluating 'physics understanding', and then attempt to determine whether algebra-physics or programming-physics instruction did better along this common metric.

Instead, I have simply accepted that programming changes the nature of what is learned – it allows for the creation of a new practice – and I have set out to explore some of the properties of this new practice. Such an analysis will never give us outright answers as to whether programming-physics or algebra-physics is a better way to teach physics, although it can

inform instructional decisions. All it will tell us is how they differ, along some dimensions.

Furthermore, I believe that we are increasingly in need of this type of analysis. More and more, researchers are not trying to develop new techniques for reaching the usual instructional goals. They are, instead, often producing dramatically new images for instruction. This is this case for much of the work that the Logo community has done in the last few decades within the domain of mathematics. For example, Abelson and di Sessa (1980) proposed a radically reformulated version of geometry based around turtle graphics in Logo. In addition, Wilensky (1993) described a new vision for mathematics education called "Connected Mathematics," which is, in part, rooted in the notion that programming can play the role of an expressive medium for mathematics. Finally, Noss and Hoyles (1996) presented an extended discussion of their new vision of mathematics learning.

When we propose to radically transform a domain, simple comparative evaluations will not do; we cannot merely ask which of two instructional practices allows students to perform better along a common measure. When we define new domains or, at least, define very different modes of knowing a domain, we need techniques for understanding and comparing the alternative practices created.

There are probably many such techniques one could employ. For example, one might look to compare the skills that alternative instructional practices produce; we could ask what variety of tasks a successful student will be able to undertake following a particular variety of instruction. In this paper, I have presented a different technique, tuned to answer what I believe are some very important questions. I have been concerned with characterizing the deep nature of the understanding of the physical world associated with programming-physics and algebra-physics.

ACKNOWLEDGEMENTS

I owe a great debt to Andrea di Sessa for providing the intellectual foundations of this work and supporting its development. In addition, many people commented on the dissertation on which this work is based. Thanks, especially, to Andrea di Sessa, Miriam Gamoran, Rogers Hall, and Barbara White. Finally, I would like to thank the two anonymous reviewers for their detailed and helpful comments. This work was supported, in part, by grant number B-1393 to A. di Sessa from the Spencer Foundation.

APPENDIX A. KEY TO TRANSCRIPTS

(. . .) Text in parentheses indicates transcription uncertain.
// Start of overlap with another speaker's utterance.
::: Previous sound is drawn out.
„ Speaker trails off without finishing statement.
– Word is cut off.
= Indicates that there is not the usual amount of silence between two utterances.
(0.0) A pause, with approximate time in seconds.
[. . .] Non-linguistic act.

APPENDIX B. ALGEBRA TASKS

1. Shoved block	A person gives a block a shove so that it slides across a table and then comes to rest. Talk about the forces and what's happening. How does the situation differ if the block is heavier?
2. Vertical pitch	(a) Suppose a pitcher throws a baseball straight up at 100 mph. Ignoring air resistance, how high does it go? (b) How long does it take to reach that height?
3. Air resistance	For this problem, imagine that two objects are dropped from a great height. These two objects are identical in size and shape, but one object has twice the mass of the other object. Because of air resistance, both objects eventually reach terminal velocity. (a) Compare the terminal velocities of the two objects. Are their terminal velocities the same? Is the terminal velocity of one object twice as large as the terminal velocity of the other? (Hint: Keep in mind that a steel ball falls more quickly than an identically shaped paper ball in the presence of air resistance.) (b) Suppose that there was a wind blowing straight up when the objects were dropped, how would your answer differ? What if the wind was blowing straight down?
4. Mass on a spring	A mass hangs from a spring attached to the ceiling. How does the equilibrium position of the mass depend upon the spring constant, k, and the mass, m?
5. Stranded skater	Peggy Fleming (a one-time famous figure skater) is stuck on a patch of frictionless ice. Cleverly, she takes off one of her ice skates and throws it as hard as she can. (a) Roughly, how far does she travel? (b) Roughly, how fast does she travel?
6. Buoyant cube	An ice cube, with edges of length L, is placed in a large container of water. How far below the surface does the cube sink? (ρ_{ice} = 0.92 g/cm^3; ρ_{water} = 1 g/cm^3).
7. Running in the rain	Suppose that you need to cross the street during a steady downpour and you don't have an umbrella. Is it better to walk or run across the street? Make a simple computation, assuming that you're shaped like a tall rectangular crate. Also, you can assume that the rain is falling straight down. Would it affect your result if the rain was falling at an angle?

APPENDIX C. PROGRAMMING TASKS

1. Shoved block	Imagine that a block is resting on a table with friction. Program a simulation for the case in which the block is given a short, hard shove. How does the motion differ if the block is heavier? Can you show this in your simulation? Modify your simulation if necessary.
2. Dropped ball	The idea here is to make a realistic simulation of the motion of a ball that is dropped from someone's hand. Using the tick model, redo your simulation of a dropped ball.
3. Tossed ball	Now we want to imagine a situation in which a ball is tossed exactly straight up and then falls back down to where it started. Using the tick model, redo your simulation of a ball tossed straight up.
4. Air resistance	(a) Make a new simulation of the DROP that includes air resistance. (Hint: Air resistance is bigger if you're moving faster.) (b) Try running your simulation for a very long time. (The ball might go off the bottom and come back in the top, that's okay.) If you've done you're simulation correctly, the speed of the ball should eventually reach a constant value. (The "terminal velocity.")
5. More air resistance	Your simulation of a ball dropped with air resistance is included in this box. (a) How does the terminal velocity of a dropped ball depend upon the mass of the ball? For example, what happens to the terminal velocity if the mass doubles? Can you show this with your simulation? Modify your simulation if necessary. (b) Suppose there was a wind blowing straight up. How would your simulation and the terminal velocity be different? (c) What if the wind is blowing straight down?
6. Mass on as spring	In this task, you must imagine that the block shown is attached to a spring so that it oscillates back and forth across the surface of a frictionless table. (a) Make a simulation of this motion. (b) If the force, mass, and spring constant don't appear in your simulation, then add them.

NOTES

[1] Refer to Appendix A for a key to the notations used in transcripts.

[2] The equation here actually applies for all times following the moment when terminal velocity is reached. For this reason, it may not be immediately clear to the reader that this instance should be categorized as SPECIFIC MOMENT, rather than STEADY STATE. This type of subtlety is discussed in Sherin (1996). However, it is not of great significance for the arguments to be made in this paper.

[3] My interpretation of this episode, as well as of the episodes that follow, is at odds with many other accounts of physics problem solving. For example, many other researchers account for the construction of equations of this sort by an appeal to knowledge that is more closely tied to physical principles (e.g., Chi et al., 1981; Larkin, 1983). However, I will not attempt to present an argument for my position here. The interested reader is referred to Sherin (in press) where this argument is made in detail.

[4] Note that the PROP+ form also does not specify precisely what expression to write; it only says to write an expression that is proportional to the quantity in question. This

ambiguity is a fundamental feature of symbolic forms. Forms specify the structure of an expression only at a certain level of detail.

REFERENCES

Abelson, H. and di Sessa, A. (1980). *Turtle Geometry*. Cambridge, MA: MIT Press.

Bork, A. M. (1967). *Fortran for Physics*. Reading, MA: Addison-Wesley.

Bruner, J. S. (1966). On cognitive growth. In J. S. Bruner, R. R. Olver and P. M. Greenfield (Eds), *Studies in Cognitive Growth II* (pp. 30–67). New York: John Wiley and Sons.

Carpenter, T. P. and Moser, J. M. (1983). The development of addition and subtraction problem-solving skills. In T. P. Carpenter, J. M. Moser and T. A. Romberg (Eds), *Addition and Subtraction: A Cognitive Perspective* (pp. 9–24). Hillsdale, NJ: Erlbaum.

Chi, M. T. H., Feltovich, P. J. and Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science* 5: 121–152.

di Sessa, A. A. (1984). Phenomenology and the evolution of intuition. In D. Gentner and A. Stevens (Eds), *Mental Models*. Hillsdale, NJ: Lawrence Erlbaum.

di Sessa, A. A. (1986). From logo to boxer. *Australian Educational Computing* 1(1): 8–15.

di Sessa, A. A. (1989). A child's science of motion: Overview and first results. In U. Leron and N. Krumholtz (Eds), *Proceedings of the the Fourth International Conference for Logo and Mathematics* (pp. 211–231). Haifa, Israel.

di Sessa, A. A. (1993). Toward an epistemology of physics. *Cognition and Instruction* 10(2 and 3): 165–255.

di Sessa, A. A. (2000). *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA: MIT Press.

di Sessa, A. A. and Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM* 29(9): 859–868.

di Sessa, A. A., Abelson, H. and Ploger, D. (1991). An overview of Boxer. *Journal of Mathematical Behavior* 10(1): 3–15.

Ehrlich, K. and Soloway, E. (1984). An empirical investigation of tacit plan knowledge in programming. In J. C. Thomas and M. L. Schneider (Eds), *Human Factors in Computer Systems* (pp. 113–133). Norwood, NJ: Ablex.

Ehrlich, R. (1973). *Physics and Computers: Problems, Simulations, and Data Analysis*. Boston: Houghton Mifflin.

Feynman, R. (1965). *The Character of Physical Law*. Cambridge, MA: MIT Press.

Feynman, R. P., Leighton, R. P. and Sands, M. (1963). *Lectures on Physics*. Reading, MA: Addison-Wesley.

Forbus, K. D. (1984). Qualitative process theory. *Artificial Intelligence* 24: 85–168.

Fuson, K. C. (1992). Research on whole number addition and subtraction. In D. A. Grouws (Ed.), *Handbook of Research on Mathematics Teaching and Learning* (pp. 243–275). New York: Macmillan Publishing Company.

Goody, J. (1977). *The Domestication of the Savage Mind*. New York: Cambridge University Press.

Goody, J. and Watt, I. (1968). The consequences of literacy. In J. Goody (Ed.), *Literacy in Traditional Societies* (pp. 304–345). Cambridge, England: Cambridge University Press.

Gould, H. and Tobochnik, J. (1988). *An Introduction to Computer Simulation Methods*. Reading, MA: Addison-Wesley.

Hayes, P. J. (1979). The naive physics manifesto. In D. Michie (Ed.), *Expert Systems in the Micro-Electronic Age* (pp. 242–270). Edinburgh, Scotland: Edinburgh University Press.

Hayes, P. J. (1984). The second naive physics manifesto. In J. Hobbs (Ed.), *Formal Theories of the Commonsense World* (pp. 1–36). Hillsdale, NJ: Ablex.

Herscovics, N. and Kieran, C. (1980). Constructing meaning for the concept of equation. *Mathematics Teacher* 73(8): 572–580.

Horwich, P. (1987). *Asymmetries in Time*. Cambridge, MA: MIT Press.

Johnson, M. (1987). *The Body in the Mind: The Bodily Basis of Meaning, Imagination, and Reason*. Chicago: University of Chicago Press.

Kieran, C. (1992). The learning and teaching of school algebra. In D. A. Grouws (Ed.), *Handbook of Research on Mathematics Teaching and Learning* (pp. 390–419). New York: Macmillan.

de Kleer, J. and Brown, J. S. (1984). A qualitative physics based on confluences. *Artificial Intelligence* 24: 7–83.

Larkin, J. (1983). The role of problem representation in physics. In D. Gentner and A. Stevens (Eds), *Mental Models* (pp. 75–98). Hillsdale, NJ: Erlbaum.

Larkin, J., McDermott, J., Simon, D. P. and Simon, H. A. (1980). Expert and novice performance in solving physics problems. *Science* 208: 1335–1342.

Letovsky, S., Pinto, J., Lampert, R. and Soloway, E. (1987). A cognitive analysis of a code inspection. In G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop* (pp. 231–247). Norwood, NJ: Ablex.

Linn, M. C., Sloane, K. D. and Clancy, M. J. (1987). Ideal and actual outcomes from precollege Pascal instruction. *Journal of Research in Science Teaching* 24(5): 467–490.

MacDonald, W. M., Redish, E. F. and Wilson, J. M. (1988). The M.U.P.P.E.T. manifesto. *Computers in Physics* 2: 23–30.

Mann, L. M. (1991). The implications of functional and structural knowledge representations for novice programmers. Unpublished dissertation manuscript. University of California, Berkeley.

McClosky, M. (1984). Naive theories of motion. In D. Gentner and A. Stevens (Eds), *Mental Models* (pp. 289–324). Hillsdale, NJ: Erlbaum.

Misner, C. W. and Cooney, P., J. (1991). *Spreadsheet Physics*. Reading: MA: Addison-Wesley.

Noss, R. and Hoyles, C. (1996). *Windows on Mathematical Meanings: Learning Cultures and Computers*. Dordrecht: Kluwer Academic Publishers.

Olson, D. R. (1994). *The World on Paper*. New York: Cambridge University Press.

Papert, S. (1980). *Mindstorms*. New York: Basic Books.

Redish, E. F. and Risley, J. S. (1988). *The Conference on Computers in Physics Instruction: Proceedings*. Redwood City, CA: Addison-Wesley.

Redish, E. F. and Wilson, J. M. (1993). Student programming in the introductory physics course: M.U.P.P.E.T. *Am. J. Phys.* 61: 222–232.

Reed, S. K. (1998). *Word Problems: Research and Curriculum Reform*. Mahwah, NJ: Erlbaum.

Repenning, A. (1993). Agentsheets: A tool for building domain-oriented dynamic, visual environments. Unpublished dissertation manuscript. University of Colorado.

Resnick, M. (1994). *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*. Cambridge, MA: MIT Press.

Riley, M. S., Greeno, J. G. and Heller, J. I. (1983). Development of children's problem-solving ability in arithmetic. In H. P. Ginsberg (Ed.), *The Development of Mathematical Thinking* (pp. 153–196), New York: Academic Press.

Scribner, S. and Cole, M. (1981). *The Psychology of Literacy*. Cambridge, MA: Harvard University Press.

Sfard, A. (1987). Two conceptions of mathematical notions: Operational and structural. In J. C. Bergeron, N. Herscovics and C. Kieran (Eds), *Proceedings of the Eleventh International Conference for the Psychology of Mathematics Education* (pp. 162–169). Montreal, Canada.

Sfard, A. (1991). On the dual nature of mathematical conceptions. *Educational Studies in Mathematics* 22: 1–36.

Sherin, B. (1996). The symbolic basis of physical intuition: A study of two symbol systems in physics instruction. Unpublished dissertation manuscript. UC Berkeley.

Sherin, B. (in press). How students understand physics equations. *Cognition and Instruction*.

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

Soloway, E. (1986). Learning to program = Learning to construct mechanisms and explanations. *Communications of the ACM* 29(9): 850–858.

Vygotsky, L. (1986). *Thought and Language*. Cambridge, MA: MIT Press.

Whorf, B. L. (1956). *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf*. Cambridge, MA: MIT Press.

Wilensky, U. (1993). Connected mathematics – building concrete relationships with mathematical knowledge. Unpublished dissertation manuscript. MIT.

Wilensky, U. (1997). What is normal anyway? Therapy for epistemological anxiety. *Educational Studies in Mathematics* 33(2): 171–202.

Wilensky, U. (1999). GasLab-an extensible modeling toolkit for exploring micro- and macro- views of gases. In N. Roberts, W. Feurzeig and B. Hunter (Eds), *Computer Modeling and Simulation in Science Education*. Berlin: Springer Verlag.

School of Education and Social Policy
Northwestern University
2115 N. Campus Drive
Evanston, IL 60208-2610
E-mail: bsherin@northwestern.edu