An Experimental Study of the Impact of Feedback Self-Selection on Novice Programming

(Submitted November 8, 2005)

(Revised June, 2006)

CHRISTOPHER D. HUNDHAUSEN & JONATHAN LEE BROWN
Visualization and End user Programming Laboratory
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752

Please address correspondence to

Christopher D. Hundhausen
School of Electrical Engineering and Computer Science
Washington State University
P.O. Box 642752
Pullman, WA 99164-2752
Phone: (509) 335-4590
Fax: (509) 335-3818
hundhaus@eecs.wsu.edu

Abstract

Prior empirical studies of programming have shown that novice programmers tend to program by exploration, relying on frequent compilation and execution of their code in order to make progress. One way a programming environment might facilitate this exploratory programming process is to support a “live” editing model, in which immediate feedback on a program’s syntactic and semantic correctness is provided automatically at edit-time. Notice that the notion of “liveness” actually encompasses two distinct dimensions: (a) the amount of time a programmer must wait between editing a program and receiving syntactic and semantic feedback (*feedback delay*); and (b) whether such feedback is provided automatically, or whether the programmer must explicitly request it (*feedback self-selection*). While a few prior empirical studies of “live” editing do exist, none has specifically evaluated the impact of these dimensions of “live” editing within the context of the imperative programming paradigm commonly taught in first-semester computer science courses. As a preliminary step toward that end, we conducted an experimental study that investigated the impact of *feedback self-selection* on novice imperative programming. Our within-subjects design compared the impact of three different levels of feedback self-selection on syntactic and semantic correctness: (a) no syntactic or semantic feedback at all (the *No Feedback* treatment); (b) syntactic and visual semantic feedback immediately provided on request when a “run” button is hit (the *Self-Select* treatment); and (c) syntactic and visual semantic feedback immediately provided on every keystroke (the *Automatic* treatment). The *Automatic* and *Self-Select* treatments yielded programs that had significantly fewer syntactic and semantic errors than those of the *No Feedback* treatment; however, no significant differences were found between the *Automatic* and *Self-Select* treatments. These results suggest that, at least within the context of novice imperative programming, the value of “liveness,” if there is any, may stem from minimal feedback delay.

1. Introduction

Computer programming environments differ with respect to the times at which, and the delay with which, they furnish syntactic and semantic feedback to programmers. For example, in the case of a traditional text editor coupled with a compiler, the programmer must explicitly request syntactic feedback by executing the compiler. Likewise, the programmer must explicitly request semantic feedback by executing the program itself. This traditional edit-compile-execute cycle introduces a potentially substantial delay between the time a program is edited, and the time feedback on that program is obtained.

Through their support of syntax highlighting and pop-up menus containing lists of valid next tokens, modern imperative programming environments such as Eclipse® and Visual Studio® narrow the gap between program editing and syntactic feedback. Imperative programming environments geared toward novices go even further by providing visual semantic feedback on program execution (see, e.g.,[1, 2]) and program structure (see, e.g.,[3, 4]); however, because compiled languages underlie most of these environments, they still require programmers to explicitly run a program in order to receive semantic feedback.

Because they are able to execute code on the fly, a large number of modern programming environments are able to eliminate the gap between program editing and semantic feedback that is common in imperative programming environments. For example, interpreted languages such as Python [5] and Scheme [6] are able to execute on a line-by-line basis, thus providing semantic feedback at edit-time. Similarly, in declarative programming environments such as Forms/3 [7], all formulas are automatically recalculated whenever a formula is changed. Thus, users are provided with instantaneous syntactic and semantic feedback at edit-time.

Over a decade ago, Tanimoto [8] introduced the notion of "liveness" in order to characterize the level of responsiveness of a programming environment to the programmer's edits. According to Tanimoto's "liveness" spectrum, the least "live" programming environments are those that do not support program execution at all. In contrast, the most "live" programming environments are those that allow a program to be edited *while* it is executing, with any changes immediately reflected in the execution. Notice that Tanimoto's notion of programming environment "liveness" actually encompasses two distinct dimensions:

(a) the amount of time a programmer must wait between editing a program and receiving syntactic and semantic feedback (*feedback delay*); and

(b) whether such feedback is provided automatically, or whether the programmer must explicitly request it (*feedback self-selection*).

While a few prior empirical studies of the impact of programming environment "liveness" on programming do exist, it is noteworthy that no previous studies have specifically evaluated the impact of the above dimensions of "liveness" within the context of novices first learning the imperative programming paradigm commonly taught in college computer science courses. The following research question thus arises:

Research Question: *What is the impact of feedback delay and feedback self-selection on novice programming?*

Three well-known models and theories of human-computer interaction—Norman's *seven stages of action model* [9], Shneiderman's principle of *direct manipulation* [10], and Green and Petre's 'cognitive dimension' of *progressive evaluation* [11]—all reach a similar conclusion with respect to the value of automatic, continuous feedback on task performance: In order for users to be successful, they must be able to *continuously* evaluate their progress toward their goals. Hence, a programming environment with continuously updated (automatic) feedback ought to promote more accurate programming than one that does not provide continuously updated feedback. This suggests the following hypothesis:

H1: *Continuous edit-time feedback will promote the development of programs that are more correct than those promoted by feedback that is provided only on request.*

In contrast, based on educational psychology's cognitive load theory (see, e.g., [12]), one might posit that automatic feedback may not have any advantages over on-request feedback in a programming task. Notice that, in order to benefit from automatic feedback during the act of programming, the programmer must mentally integrate that feedback with the evolving algorithm source code. According to cognitive load theory, such integration imposes a potentially substantial *extraneous cognitive load*. Compounding this extraneous cognitive load is the *intrinsic cognitive load* brought about by the programmer's need to shift from the local editing context (a line of code) to a more global context (the *programming plan* [13] of which the current line is a part) in order to make sense of semantic feedback. This line of reasoning suggests the following opposing hypothesis:

H2: Automatic feedback will promote the development of programs that are equally as correct as the programs promoted by feedback that is provided on request.

In addition to helping to resolve the dilemma raised by these opposing hypotheses, experimental studies of novice programming environment "liveness" are needed to guide the future evolution of novice programming environments, which have chosen to adopt features that support increasing levels of "liveness," in spite of the absence of any empirical evidence of its benefits.

As an initial step toward these ends, this article presents an experimental study that systematically investigated the impact of one dimension of programming environment "liveness"—syntactic and semantic feedback self-selection— on novice programming performance. Our experiment adopted a within-subjects design that compared three different levels of syntactic and semantic feedback:

(a) no syntactic or semantic feedback at all (the *No Feedback* treatment);

(b) syntactic and visual semantic feedback provided on demand when a "run" button is hit (the *Self-Select* treatment); and

(c) syntactic and visual semantic feedback updated on every keystroke (the *Automatic* treatment).

We found that the *Self-Select* and *Automatic* treatments promoted the development of programs with significantly fewer syntactic and semantic errors than those promoted by the *No Feedback* treatment. However, we found no significant differences between the *Self-Select* and *Automatic* treatments with respect to either syntactic or semantic correctness. We must therefore conclude that, within the context of novice imperative programming tasks, the benefits of "liveness," if there are any, stem from minimal feedback delay.

The remainder of this article is organized as follows. In Section 2, we review related work. Section 3 details the design and methods of our experiment, while Sections 4 and 5 present and discuss our results. We conclude, in Section 6, by summarizing our findings, considering their implications for programming environment design, and identifying directions for future research.

2. Related Work

As noted above, a great deal of past research has been concerned with the development of novel programming environment interfaces that enable programmers to complete tasks more quickly and efficiently than they can with conventional text editors coupled with compilers. Dating from the late 1970s, one line of this research has explored the idea of structure editors (e.g., [14-16], which prevent syntax errors by providing an interface through which programmers can edit a program's syntax tree directly. Insofar as it explores user interface mechanisms for enabling users to create syntactically correct programs more quickly, this line of work marginally relates to our interest in the issue of feedback self-selection. However, very little of this work (cf. [17]) has been rooted in, or subjected to, empirical studies, making it difficult to draw definitive conclusions on the efficacy of its approach.

Within the general area of human-computer interaction, much research has explored the question of how software can best provide feedback to the user. Notice that answers to this question might address at least three different aspects of feedback, the last two of which are embodied in Tanimoto's [8] notion of "liveness":

(1) *Feedback content*—What particular information do users need in order to make progress toward their goals at a particular point in a task sequence?

(2) *Feedback delay*—How quickly do users need feedback in order to perform tasks quickly and accurately?

(3) *Feedback self-selection*— Should the software provide feedback automatically and continuously, or only when the user requests it?

In the remainder of this section, we review related empirical studies that considered each of these three aspects of feedback.

2.1 Feedback Content Studies

Clearly, "live" feedback will be of little value if users cannot meaningfully interpret it to identify and recover from errors, confirm successes, and ultimately make progress toward their goals. Several studies of novice programming have shed light on the content of effective textual feedback messages. In a study of a Lisp intelligent tutoring system, Corbett and Anderson [18] compared feedback messages that explained why a piece of code was incorrect against feedback messages that simply reported that an error had occurred. The results showed that students who received explanatory feedback made significantly fewer intermediate programming errors, although there were no significant differences with respect to final code correctness or time-on-task.

In a similar vein, Shneiderman [19] considered the impact of differing COBOL error messages on novice programming performance in a series of six studies. A general result of Shneiderman's studies was that specific error messages that explained how to fix the problem not only improved error recovery rates, but also garnered higher subjective ratings.

While the syntactic feedback in our study was textual, the *semantic* feedback provided by the programming environments used in our study came in the form of visualizations of program variables and arrays. These visualizations corresponded to the state of the algorithm at the current execution point, marked by an arrow in an algorithm source code window. Prior research into the use of complementary textual and pictorial representations as learning aids (such as the code and visualization windows in our study's programming environments) has shown that learners may have trouble benefiting from these multiple representations unless they are able to integrate the two representations into a coherent mental model (see, e.g., [20, 21]). According to one line of empirical research, requiring learners to actively draw connections between multiple representations before actually using those representations as learning aids can help learners to integrate the representations, leading to better learning outcomes [22].

2.2 Feedback Delay Studies

The majority of computer feedback studies have focused on the issue of how long a computer system should take to provide feedback on user actions. Reviewed in chapter 10 of [23], this corpus of studies has yielded somewhat inconsistent results. In complex problem-solving tasks, users were capable of adapting their work strategies to the rate of feedback. However, there was no clear benefit to shorter feedback delay times with respect to both task efficiency and accuracy. For example, two different studies of complex problem solving tasks (matrix multiplication and statistics) [24, 25] found that feedback delay times that ranged from as low as 0.1 seconds to as high as 64 seconds resulted in no differences with respect to the time participants needed to produce correct solutions. In a study of circuit layout tasks by Barber and Lucas [26], however, a feedback delay "sweet spot" of 12 seconds was found, even though participants subjectively preferred shorter feedback delays.

A key difference between the study presented here and this line of studies is that we considered novice performance, whereas these prior studies considered expert performance. In addition, in contrast to these prior studies, the feedback delay in the two feedback treatments in our study was fixed; it was set to be essentially instantaneous. Thus, while the results of this prior line of work might predict user satisfaction to be equally high in both of our feedback treatments, this prior line of work cannot make any predictions regarding user performance.

Building on these prior studies of feedback delay, a more recent study by Wilcox *et al.* [27] shares our study's goal of explicitly investigating of the impact of programming environment "liveness" on task performance. In the

Wilcox *et al.* study, senior-level computer science students performed two debugging tasks using "live" and "non-live" versions of a (declarative) spreadsheet environment [7]. The "live" version provided syntactic and semantic feedback immediately after a cell was edited and dismissed, whereas the "non-live" version provided syntactic and semantic feedback 90 seconds after it was explicitly requested with a press of a "run" button. The study found that the average number of bugs corrected with the help of each alternative environment varied by problem, with the "live" environment yielding a significant advantage for one of the problems. A follow-up analysis of the study data by Cook *et al.* [28] found that, with respect to those bugs that were ultimately corrected, participants in the "live" condition performed the correction significantly faster than did participants in the "non-live" condition.

The study presented here differs from that of Wilcox *et al.* in four key respects. First, our study focused on *novice* programming activity, as opposed to the more expert programming activity of senior-level computer science students. Second, whereas the Wilcox *et al.* study looked at *declarative* (spreadsheet) programming, our study considered the kind of *imperative* programming that students typically learn in an introductory computer science course. Third, we believe a potential confound in the Wilcox *et al.* study was the fact that its two experimental conditions actually varied with respect to two separate factors: *feedback delay* ("live"—0 seconds; "non-live"—90 seconds) and *feedback self-selection* ("live"—no; "non-live"—yes). Our study attempted to disentangle these two factors by fixing feedback delay and varying feedback self-selection. Fourth, and perhaps most notable, our study investigated the impact of "liveness" on writing a program from scratch, as opposed to debugging a program that has already been written and deliberately injected with bugs.

This fourth difference can be seen as a key extension to this previous work. Indeed, in their own discussion of their results, Wilcox *et al.* point out that an "as-yet open question is whether "liveness" helps keep the bugs out during initial program construction" (p. 264); they go on to identify this question as "one for which many developers have strong intuitions that need to be either debunked or verified" (p. 264).

2.3 Feedback Self-Selection Studies

Besides the study presented here, we are aware of only one other series of studies that has examined the benefits of providing feedback automatically versus only on request. As part of a large body of studies of various versions of the Lisp Intelligent Tutoring System, Corbett and Anderson considered, in two separate studies [29, 30], the question of feedback *timing*: Should feedback always appear whenever an error is detected or the learner appears to be struggling, or should feedback appear only when the learner explicitly requests it? In both of these studies, no differences in programming accuracy were detected between a treatment group that obtained feedback only on request, and a treatment group that automatically received feedback whenever the system detected that the user was struggling, or whenever an error was detected. However, the treatment group that received automatic feedback performed programming tasks significantly faster.

Like the studies of Corbett and Anderson, the study presented here focused on the impact of feedback self selection on novice programming. There are, however, two notable differences between Corbett and Anderson's studies and our study. First, our study considered programming in a simple imperative pseudocode-like language, rather than the functional Lisp language. Second, unlike the intelligent tutoring system considered by Corbett and Anderson, our experimental software was merely a programming environment; it was not a tutor. Thus, it was unable to provide feedback that guided the user's programming efforts according to a hierarchical plan of programming goals; it was unable to provide feedback when it sensed that the user was struggling; and it was unable to provide feedback that prevented the user from straying from an ideal solution. Clearly, an ability to provide these three forms of feedback places an intelligent tutoring system on unequal footing with a novice programming environment. It is therefore difficult to make predictions about the results of our study based on Corbett and Anderson's results.

3. Design and Methods

The main objective of the experiment presented here was to investigate systematically the impact of feedback self-selection on novice programming performance within the imperative programming paradigm commonly taught in college computer science courses. To that end, we adopted a single-factor within-subjects design in which each participant was exposed to all of the following three levels of the *feedback self-selection* independent variable:

1. *No feedback*. This is akin to asking someone (a) to write a program in a text editor or with pen and paper, and then (b) to mentally simulate the program on a sample input data set. Because it promotes active mental simulation of program execution, such an exercise may well have pedagogical benefits; however, it also requires that the programmer have, a priori, a serviceable mental model of program execution—an ambitious expectation for novices just learning to program. Although novice

programmers may only seldom program in this way, we included this level of feedback immediacy as a control treatment against which to compare the other levels of feedback self-selection.

2. *Self-Select feedback*. The user must explicitly request syntactic and semantic feedback. This is the normal state of affairs in novice programming and visualization environments such as ALICE [1], Jeliot [2], BlueJ [3], and Raptor [4]. Although some of these environments do support automatic feedback on syntactic correctness, programmers must still explicitly hit a run button in order to obtain visual execution feedback.
3. *Automatic feedback*. While common in declarative programming environments, this is a level of feedback that, because it poses implementation challenges, is rare in imperative programming environments. On every keystroke, an environment supporting automatic feedback re-evaluates the program. If a syntax error is present, that error is reported in a pop-up tooltip. Otherwise, a visual representation of the program, which provides semantic feedback on its execution state, is automatically updated to reflect the current state of the program.

It is important to note two points with regard to this design. First, in the Self-Select and Automatic treatments, we held the other dimension of "liveness," *feedback delay*, constant at essentially 0 seconds: Users received feedback *immediately* after they hit the run button or typed a keystroke. (Processing requirements actually incurred a delay of up to one tenth of a second, but this was not noticeable in practice.) Second, while it would certainly be possible for a programming environment to support a level of syntactic feedback self-selection that differed from its level of semantic feedback self-selection, our experiment considered only three possible combinations: those in which syntactic and semantic feedback were provided *at the same level*. Our reasoning was that these three combinations were most likely to occur in practice, and that they best embodied the feedback differences whose effects we wanted to measure.

To measure differences between these different levels of feedback self-selection, we defined two primary dependent variables:

1. *Semantic correctness*—The percentage of semantic components from an "ideal" solution that are included in a program solution.
2. *Syntactic correctness*—The percentage of statements in a program solution that are syntactically correct.

Given that time on task proved to vary significantly in the prior feedback self-selection studies of Corbett and Anderson [29, 30], one might wonder why our design did not include task completion time as a dependent measure. In a large "pilot" version of this study, we asked participants to complete tasks as quickly as possible without sacrificing accuracy, and we put no restriction on the amount of time they could spend on a given task. It turned out that this led to a mortality confound: participants in the two feedback treatments dropped out at higher rates than participants in the no feedback treatment, because they tended to spend so much time on early tasks that they could not get to later ones. In order to avoid this confound, we opted, in this study, to fix time on task and measure accuracy alone.

3.1 Participants

Our participants were drawn from the fall, 2005 offering of CS 121 ("Program Design and Development"), the introductory computer science course at Washington State University. The course took an "algorithms-first" approach to teaching computer science. In the two weeks of instruction prior to the study, students explored algorithms and algorithmic problem solving using general pseudocode. Some 150 students, mostly freshmen and sophomores, were enrolled in the course.

We ran the study as part of the regularly-scheduled three-hour lab sessions that took place in the third week of the course. Students received course credit for participating in the study sessions. However, we did not collect data on every student who participated in the labs. Two factors influenced whether a given student was included in the study. First, we used a background questionnaire to screen students for prior programming experience. Students who self-reported any prior programming experience were excluded from the study. Second, by signing an informed consent form, students were given the option of allowing videos of their programming activities (not analyzed here), along with their final code solutions, to be collected and analyzed for the purposes of this study.

In the end, we identified 57 students (52 males and 5 females; mean age 19.8 years) in the course who were both eligible (because they had no prior programming experience) and willing (because they signed the informed consent form) to participate in the study.

3.2 Materials and Tasks

All participants worked on Pentium IV computers running the Windows XP operating system. Equipped with mice and keyboards, the computers had 1 GB of RAM and either a 15 or 18 inch LCD color display set to a resolution of 1024 × 768. Participants completed three roughly equivalent programming tasks with the help of three different programming environments, each of which supported one of the three levels of our feedback self-selection variable. Below, we describe the three alternative programming environments used by participants, the training materials they received, and the programming tasks they completed.

3.2.1 Programming Environments

The programming environments used in the three experimental treatments of this study were derived from the ALVIS Live! algorithm visualization and programming environment [31]. The ALVIS Live! environment supports programming in SALSA, an interpreted "mini language"[32] with a pseudocode-like syntax. Table 1 summarizes the SALSA language's nine valid commands, which were designed specifically to support single-procedure, array-iterative algorithms.

Figure 1 presents the version of ALVIS Live! used by participants in the Automatic condition. In this version of ALVIS Live!, the user directly types SALSA commands into the Script Window on the left. The focal point of the environment is the green Execution Arrow, which marks the line of code that was most recently executed, and that is currently being edited. On every keystroke, that line of code is re-executed. Syntactic feedback appears in a pop-up tooltip just below the line of code currently being edited (see Figure 2). Likewise, up-to-the-keystroke visual semantic feedback, in the form of a graphical depiction of the code's variables, arrays, and data movements, appears within the Animation Window on the right.

In order to facilitate automatic feedback, the execution arrow, which can also be moved around with the Execution Controls, follows the editing caret around in the Script Window. Thus, whenever the programmer moves the caret to a new line, the script is automatically executed (either forwards or backwards) to that point, and the visual representation of the program displayed in the Animation Window is updated accordingly. This "execution-follows-the-caret" behavior thus provides, on an edit-by-edit basis, dynamic syntactic and semantic feedback on the programmer's coding efforts.

Figure 3 presents a snapshot of the modified version of ALVIS Live! used by participants in the Self-Select condition. As with the version of ALVIS Live! used in the Immediate condition, users directly type commands into the Script Editor of this version of the software; however, they do not immediately see the results of those commands. Rather, they must press either the "step forward" (the blue single-arrow button in the Execution

Controls; see Figure 3) or "execute forward" (the blue double-arrow button in the Execution Controls; see Figure 3) in order to "compile" and "execute" their code. When one of those buttons is pressed, the script is "background executed" to ensure that it is syntactically correct. If syntax errors are detected, an Error Window, which appears below the Script Window (see Figure 4 for an example), reports a full list of those syntax errors. Otherwise, the text in the Script Window changes from plain text to syntax-highlighted text, and users enter "execution" mode, in which it is possible to execute the script forwards and backwards using the execution controls (just as is the case in the Automatic condition; see Figure 1). If users attempt to make an edit within "execution" mode, a dialog box appears asking them to confirm that they want to leave "execution" mode in order to make further edits to the script (see Figure 5).

Figure 6 presents a snapshot of the greatly stripped-down version of ALVIS Live! used by participants in the *No Feedback* condition. In fact, this version of ALVIS Live! is nothing more than a text editor; all of the code execution and visualization features are turned off. Users interact with this version of the software simply by typing commands into the window. Their commands, which appear in plain (unhighlighted) text, cannot be executed.

3.2.2 Training Materials

Prior to each task, participants completed a 10-minute tutorial that introduced them to the particular version of the ALVIS Live! environment that they would be using. The tutorial walked participants through a task in which they constructed and debugged a simple "Find Max" algorithm, which was similar to the algorithms they would subsequently write in the programming tasks. Upon completion of the tutorial, participants were given a one-page SALSA quick-reference guide containing a summary of each SALSA command. Participants were allowed to use the quick-reference guide throughout the programming tasks.

3.2.3 Programming Tasks

All participants completed the same three programming tasks: *compute sum*, *replace zeroes*, and *reverse list* (see Table 2). Notice that all of the tasks involved creating, populating, and iterating over one or more arrays. The main difference among the tasks lay in the manner in which array elements had to be processed. In the first task, array elements had to be summed. In the second task, all 0-valued array elements had to be replaced with the value -

1. In the third task, array elements had to be reversed through the use of a second array. Note that the task instructions emphasized that correct solutions had to be able to handle arrays of varying size.

Because we adopted a within-subjects design, we needed to ensure that the study tasks were equivalent in terms of difficulty. In order to do this, we created an “ideal” solution to each problem, and identified a set of essential semantic components embodied by each solution. Each semantic component corresponded either to a key line of code (e.g., creating an array), or to a property that a correct solution should possess (e.g., the loop should visit each array element). Our ideal *compute sum* algorithm had 11 semantic components, whereas our ideal replace zeroes and reverse list solutions had 13 semantic components. As we shall illustrate in the following section, a post-hoc statistical analysis revealed that task difficulty did not contribute to the effects we observed, thus confirming that the three tasks were sufficiently equivalent.

3.3 Procedure

In order to counterbalance treatment order, we randomly assigned participants to one of six possible treatment orders prior to running the study (see Table 3). Ideally, we would have also counterbalanced the order in which participants completed the three tasks (giving us 6 × 6 = 36 possible task-treatment orders). However, practical constraints made task counterbalancing infeasible. In particular, because we ran the study within five regularly-scheduled computer science labs, we were forced to run participants in groups of 10 to 12. In order to maximize the possibility that participants read and understood the tasks, we felt it was important to read the task instructions aloud to the groups of participants. This meant that all participants had to complete the study tasks in the same order. While we readily admit that this design flaw could have produced a confounding order or carryover effect, we confirmed through a post hoc statistical analysis (see the next section) that there was no significant sequence effects.

The general procedure for a given lab section was as follows. Prior to their lab section, participants completed a background questionnaire and informed consent form. Upon arriving at their lab section, participants were given an instruction packet corresponding to the treatment order to which they were assigned. Once all participants in a given lab section were seated, the experimenter read aloud a set of general instructions as participants followed along. Following that, participants completed the three study tasks in the same order, Prior to working on each task, participants were first given 10 minutes to work through, on their own, a tutorial that covered the experimental version of the software with which they were to complete the task; we then read aloud the instructions for the task.

Based on a pilot study, participants were given 40 minutes to complete the first task, and 35 minutes to complete each of the final two tasks. They were asked to complete the tasks as accurately as they could within the allotted time, and to use any extra time they had to go back and verify the correctness of their code. Students saved their code solutions to their local hard drives.

3.4 Code Solution Scoring Methods

To evaluate the syntactic correctness of participants' code, we developed a computer program that calculated the percentage of syntactically correct lines in a solution. We ran all participants' code through this program to automatically calculate the percentage of lines that were syntactically correct.

To evaluate the semantic correctness of participants' solutions, we checked them against the sets of semantic components in the ideal solutions we developed. Most semantic components corresponded to a single line of SALSA code. The exceptions were those components that handled the iterative aspects of the algorithm; such components were distributed over multiple statements within and including a *while* statement. As mentioned above, there were 11 semantic components in our Task 1 solution, and 13 semantic components in our Task 2 and Task 3 solutions.

Consistent with the strategy that college instructors often use to grade exams, we wanted to give credit to a solution component that was syntactically incorrect, but that captured the essence of a semantic component. To enable us to apply such a lenient grading policy consistently, we developed a detailed grading strategy, the essence of which can be summarized by the following three rules:

1. If a semantic component is internally correct, but, because of errors in the code, it is never reached, it is still counted as correct.

2. If minor syntactic changes will correct a semantic component, then it is counted as correct.

3. If a component is broken simply because a previous component is broken, but it would otherwise be correct, then that component is correct**.**

In order to ensure that our system for grading semantic correctness was reliable, we had two independent graders score 20 percent of participants' code solutions according to a more detailed version of the grading strategy outlined above. The two graders achieved an average level of agreement of 95% across all three tasks. We thus concluded that our grading rules were sufficiently reliable, and had a single grader score the remaining 80% of the code solutions. Note that, for the purpose of the statistical analyses presented in the next section, all scores were converted to percentages correct.

4. Results

Table 4 presents, by treatment, the mean percentage of syntactically correct lines, and the mean percentage of correct semantic components. Figure 7 contains box plots of the data by treatment, whereas Figure 8 contains box plots of the data by treatment order. Inspection of these data suggests that the two experimental treatments had a clear advantage over the No Feedback treatment with respect to both syntactic and semantic correctness, and that no particular treatment order appeared to be advantageous. However, given that we counterbalanced treatment order but not task order (formally called a "crossover" design), we had to be careful, in our statistical analysis of the data, to ensure that any measured differences were due to the treatment effect, and not to possible task sequence effects, or to possible carryover effects from a given task-treatment combination.

To measure such effects alongside treatment effects, we first used normal probability plots, and plots of residuals to predicted values, to verify that our data met assumptions of normality and equal variance. We then ran a general linear model analysis of variance (ANOVA) for our crossover design. The analysis found no task sequence effects or carryover effects with respect to either syntactic correctness (sequence: $df = 5$, $F = 0.44$, $p = 0.8217$; carryover: $df = 2$, $F = 1.73$, $p = 0.1827$) or semantic correctness (sequence: $df = 5$, $F = 0.48$, $p = 0.7914$; carryover: $df = 2$, $F = 0.54$, $p = 0.5823$). However, the ANOVA did find a significant treatment effect with respect to both syntactic correctness ($df = 2$, $F = 29.15$, $p < 0.0001$) and semantic correctness ($df = 2$, $F = 8.07$, $p = 0.0005$). Post-hoc Tukey-Kramer tests indicated that the differences lay between the Self-Select treatment and the No Feedback treatment (syntactic correctness: $p < 0.0001$; semantic correctness: $p = 0.0004$) and between the Automatic and No Feedback treatment (syntactic correctness: $p < 0.0001$; semantic correctness: $p = 0.0321$); however, no differences were detected between the Automatic and Self-Select treatments (syntactic correctness: $p = 0.9776$; semantic correctness: $p = 0.3620$).

Notably, our ANOVA also detected a marginally significant subject (sequence) effect with respect to syntactic correctness ($df = 51$, $F = 1.41$, $p = 0.072$), and a significant subject (sequence) effect with respect to semantic correctness ($df = 51$, $F = 8.62$, $p < 0.0001$). This latter result well reflects the wide variance typically seen in novice performance, and underscores the importance of employing within-subjects designs in studies of novice programming.

In sum, our results showed that, while the two feedback treatments yielded significantly better syntactic and semantic correctness than the No Feedback treatment, the two feedback treatments did not differ significantly from each other. Through post-hoc statistical tests, we confirmed that the observed effects were due to treatment, and not to task order, thus putting to rest any concern that our failure to counterbalance task order impacted our result. Finally, we also detected a significant subject effect, indicating that participants' task performances differed substantially from one another.

5. Discussion

Having learned valuable lessons from previously running a between-subjects "pilot" version of this experiment, we felt reasonably confident in the design of this experiment. We can cite only four possible concerns:

(a) the possibility that we introduced an order confound by not varying task order in addition to treatment order;

(b) the possibility that the three experimental tasks were not sufficiently isomorphic, which might lead to a task confound;

(c) the possibility that participants could not meaningfully interpret our textual syntactic feedback ; and

(d) the possibility that participants were unable to benefit from the visual semantic feedback because of an inability to integrate the code and visualization views into a coherent mental model (see, e.g., [20, 21]).

However, as our post-hoc statistical analysis showed, neither concern (a) nor (b) manifested itself in our results. Indeed, our general linear model ANOVA ruled out order, task, and carryover effects, and found that the only significant effects were attributable to treatment and subject, the latter of which was successfully handled by our within-subjects design. With respect to concern (c), while we cannot say for certain that all participants were able to take full advantage of our syntactic feedback messages, we made those messages as specific as possible, and made sure to offer specific suggestions for how to fix the errors, in accordance with Shneiderman's prior research on the design of textual feedback [19]. Finally, to address concern (d), we included a tutorial in the study that familiarized participants with the relationship between the code and visualization windows; however, our study did not contain the kind of active integration task recommended by prior research [22], so we cannot fully rule out the possibility that at least some of our study participants may have failed to comprehend the connections between our semantic feedback (variable and data structure visualizations) and the algorithm source code of their solutions.

Given the apparent absence of confounding factors in our results, it appears that we can take our results at face value. In practical terms, these results suggest the following:

- As long as the feedback is delivered instantaneously, automatic and self-selected feedback both yield a significant improvement in syntactic and semantic accuracy, as compared to no feedback at all.
- As long as the feedback is delivered instantaneously, both automatic and self-selected feedback are not significantly different from each other.

It is important to emphasize that our study considered just one particular set of tasks, one particular novice programming environment, and one particular form of syntactic feedback and semantic feedback. Thus, one must exercise extreme caution in any attempt to generalize the above findings beyond this set of particulars; clearly, there is no guarantee that these results will hold under different conditions.

Given our results, we must reject the first of our original hypotheses (H1), which predicted that automatic feedback would yield higher accuracy than self-selected feedback. Recall that H1 was inspired by three well-known models and principles of human-computer interaction: Norman's *seven stages of action model* [9], Shneiderman's principle of *direct manipulation* [10], and Green and Petre's 'cognitive dimension' of *progressive evaluation* [11]. All of these suggest that *continuous* feedback would be superior because it would better help users to detect and repair errors, evaluate progress toward their goals, and ultimately converge on those goals. However, in contrast to these models and principles, our results suggest that, at least within the context of novice programming, the important property of syntactic and semantic feedback is not that it be *continuous* (i.e., always visible and automatically updated), but rather that it be *immediately available* when the user wants it.

We speculate that, in less cognitively-demanding tasks, H1 would prove a viable hypothesis. In such tasks, there would appear to be a stronger need for continuous feedback, because the overhead of requesting feedback would appear to disrupt workflow, and hence have a greater impact on performance. In contrast, in more cognitively-demanding tasks such as programming, we have found that the overhead of requesting feedback is small relative to the task itself. In fact, an explicit request for feedback might even be seen as a sign of *readiness*, on the part of a user, to switch focus from the local editing context to a more global execution context.

Such speculation suggests why our second original hypothesis (H2) may have correctly predicted our results. Recall that, based on cognitive load theory, H2 predicted that there would be no difference between self-selected and automatic feedback, because taking advantage of both forms of feedback would impose an equally substantial cognitive load. Whereas H1 posited that continuous feedback would help users assess progress more effortlessly and without disruption, H2 conjectured that the very act of assessing progress with respect to a global programming plan would be disruptive within the context of an immediate editing task. Based on our results, we can speculate that the local-to-global context-switching required to integrate continuous visual execution feedback imposes such a high cognitive load that novices cannot take advantage of it. Instead, we suspect that they end up doing exactly what users of self-selected feedback do: they choose to integrate semantic feedback only when they are ready to do so. Notice that this interpretation acknowledges the real possibility that novices are, in fact, able to take advantage of automatic *syntactic* (local) feedback, because integrating such feedback would not require such context-switching; however, our experiment's dependent variables, which only considered *final* syntactic correctness, were simply not sensitive to this possibility.

Our participants' subjective impressions of automatic versus self-selected feedback shed further light on our results. In an exit questionnaire, we asked participants to rate the three environments they had used on a scale of 1 to 10 with respect to four factors: (a) helpfulness of error feedback, (b) helpfulness of programming suggestions, (c) overall helpfulness, and (d) overall programming environment effectiveness. Participants rated both the automatic and self-selected feedback environments favorably on all four measures (between 7.6 and 8.7 out of 10), with the automatic feedback environment receiving somewhat higher ratings (0.5 higher on average); however, none of the differences was statistically significant, indicating that participants found both environments to be roughly equally helpful and effective.

Interestingly, when asked directly which environment they preferred and why, 65% of participants chose the automatic feedback version, 23% chose the self-selected feedback version, and 12% had no preference. Resonating with our speculation that integrating automatic execution feedback at edit time imposes a potentially high cognitive load, seven participants expressed the concern that automatic feedback interfered with their programming at least some of the time. As one participant put it, "I preferred [self-selected feedback] because I could see how the algorithm executed when I was done, but I was not interrupted while I was writing." Likewise, another participant stated that the self-selected feedback version “allowed you to attempt to write up the code in peace,…but if there was something wrong it would help you.”

In contrast, the participants who preferred the automatic feedback version tended to cite its immediacy as being extremely valuable. Unsurprisingly, many of these participants expressed a lack of confidence in their own programming skills, and felt that the automatic feedback made it, as one participant put it, “easy to understand how the code was carried out.” This was because, as another participant put it, “you had both the code and the visualizations at the same time.” In fact, in at least one instance, automatic feedback led to an “epiphany” that helped the participant to see “how the code should be written for the desired algorithm.” A few others felt that having to explicitly request feedback was overly cumbersome; these participants felt that the convenience of automatically receiving feedback made the automatic feedback version superior.

6. Summary, Implications, and Future Work

As we have shown, many aspects of feedback, as it relates to the completion of computer-based tasks, have been empirically studied; however, few studies have looked specifically at the impact of *feedback self-selection* on task performance, and even fewer have examined the impact of feedback self-selection within the context of novice programming tasks. Moreover, relevant models, principles, and theories drawn from human-computer interaction and educational psychology lead to contradictory hypotheses regarding the impact of feedback self-selection on novice programming. Some predict that automatic feedback should lead to superior performance, while others predict that automatic feedback should be no better than on-request feedback. As a preliminary step toward resolving this dilemma within the context of novice imperative programming, and in order to provide designers of novice programming environments with empirical guidance, we have presented an experimental comparison of three different levels of feedback self-selection (automatic, self-selected, and none) on novice programming accuracy. While we found that novice programmers tend to prefer automatic feedback over self-selected feedback, our results indicate that neither automatic nor self-selected feedback promotes higher syntactic and semantic accuracy, as long as the feedback is delivered without delay.

Our results have important implications for the design of visual programming environments. From an implementation standpoint, providing continuously-updated, automatic feedback is more challenging than providing feedback that is provided on-request. Our results provide the beginnings of an empirical case against the need for *automatic* feedback—one aspect of "liveness" [8], a programming environment feature that has been highly touted in the visual languages community. In particular, our results suggest that, rather than coming up with ways to facilitate "liveness," programming environment designers ought to be putting their efforts into designing *effective* semantic feedback that benefits users. This involves carefully considering *when* users will be ready to take advantage of the feedback that is coming toward them, along with *what* content will be most beneficial.

In addition to its implications for programming environment design, the study presented here raises several questions for future research. We are particularly interested in pursuing three of these. First, recall that a key finding of our study is that novices must be "ready" in order to benefit from semantic feedback. We are unaware of past psychological research that has specifically explored the idea of "readiness" within the context of novice programming. We believe that studies that investigate this issue would be a valuable contribution to the literature We can imagine instrumenting the ALVIS software such that it loggged a current code snapshot immediately before each execution attempt. Analyzing the progression of such code snapshots could provide insight into the following research questions:

- At what points do novices typically request feedback?
- Are there particular patterns of requests that lead to higher semantic accuracy?
- Can we design "smart" feedback mechanisms that step in and provide feedback when users fail to request feedback at times when it would likely help them?

Second, recall that, as an initial step, our study chose to examine just one of the two dimensions of Tanimoto's notion of "liveness" [8]. The other dimension of "liveness," *feedback delay,* was fixed at essentially 0 seconds. If it is true, as past programming environment research appears to believe, that "liveness" is an important attribute of programming environments, then *feedback delay* must be the key factor, at least for novice programming environments. In an upcoming experimental study, we plan to investigate the impact of feedback delay on programming outcomes by systematically varying feedback delay at regular intervals between 0 and 30 seconds. Such a study will shed even more light on the value of "liveness" in novice programming, and perhaps provide the empirical case for "liveness" that is presently lacking.

Second, in choosing to focus primarily on syntactic and syntactic correctness as dependent measures, our study, as presented here, provides but a narrow glimpse of the impact of feedback self-selection on novice programming. Indeed, the data we reported here say little about the ways in which differing levels of feedback self-selection impacted the *programming processes* by which participants arrived at their solutions. In both this study and the "pilot" run of this study that was mentioned above, we recorded lossless videos of all participants' coding sessions. Based on an iterative analysis of a sample of the video collected in our "pilot" study, we have already devised a principled scheme for coding the video data of participants' programming sessions, and for transforming those codes into a set of quantitative measures that can shed substantial light on the programming process differences promoted by differing levels of feedback. A strength of our coding scheme is that it allows us to generate timeline visualizations that illustrate (a) how participants direct their coding efforts over time, (b) when participants request feedback, and (c) how feedback ultimately impacts their coding efforts. Using our scheme, three independent coders, with one month of training, achieved 95% agreement on a sample of our video data from the "pilot" run of this study. They proceeded to code all of the Task 1 videos gathered from that pilot study (nearly 33 hours of recordings). In an upcoming publication, we look forward to presenting our coding methodology and the results of this post-hoc video analysis, which will complement and shed further light on the results presented here.

7. Acknowledgments

Sean Farley and Sudhir Garg helped implement the ALVIS LIVE! environment used in the experiment. Sean Farley assisted us greatly in preparing for and carrying out this study. Dr. Nairanjana Dasgupta, associate professor of statistics at Washington State University, designed and carried out the statistical analyses presented in Section 4. This research is funded by the National Science Foundation under grant no. 0406485.

8. References

1. W. Dann, S. Cooper, & R. Pausch (2000) Making the connection: Programming with animated small world. In: *Proc. ITiCSE 2000* ACM Press, New York, pp. 41-44.
2. R. Ben-Bassat Levy, M. Ben-Ari, & P. Uronen (2003) The Jeliot 2000 program animation system. *Computers & Education* **40,** 1-15.
3. M. Kölling, B. Quig, A. Patterson, & J. Rosenberg (2003) The BlueJ system and its pedagogy. *Journal of Compuer Science Education* **13,** 249-268.
4. M. Carlisle, T. Wilson, J. Humphrieis, & S. Hadfield (2005) RAPTOR: A visual programming environment for teaching algorithmic problem solving. In: *Proc. ACM SIGCSE 2005 Symposium* ACM Press, New York, pp. 176-180.
5. W. Chun (2006) *Core Python programming* Prentice Hall, Upper Saddle River, NJ.
6. H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. S. Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, & M. Wand (1998) Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* **11,** 7-105.

7. M. Burnett & A. Ambler (1994) Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing* **5,** 29-60.
8. S. L. Tanimoto (1990) VIVA: A visual language for image processing. *Journal of Visual Languages and Computing* **1,** 127-139.
9. D. A. Norman (1990) *The Design of Everyday Things* Doubleday, New York, 257.
10. B. Shneiderman (1983) Direct manipulation: a step beyond programming languages. *IEEE Computer* **16,** 57-69.
11. T. R. G. Green & M. Petre (1996) Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Langauges and Computing* **7,** 131-174.
12. J. J. G. van Merrienboer & J. Sweller (2005) Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review* **17,** 147-177.
13. E. Soloway & K. Ehrlich (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* **SE-10,** 595-609.
14. P. Miller, J. Pane, G. Meter, & S. Vorthmann (1994) Evolution of novice programming environments: the structure editors of Carnegie Mellon University. *Interactive Learning Environments* **4,** 140-158.
15. T. Teitelbaum & T. Reps (1981) The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM* **24,** 563-573.
16. C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, & R. Pausch, "ALICE2: Programming without syntax errors," presented at User Interface Software and Technology, Paris, France, 2002.
17. A. Ko, H. Aung, & B. Myers (2005) Design requirements for more flexible structured editors from a study of programmer's text editing. In: *CHI '05 extended abstracts on human factors in computing systems* ACM Press, New York, pp. 1557-1560.
18. A. T. Corbett & J. R. Anderson (1992) The LISP intelligent tutoring system: Research in skill acquisition. In: *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration* (J. Larkin, R. Chabay, and C. Scheftic, Eds.) Erlbaum, Hillsdale, NJ, pp. 73-110.
19. B. Shneiderman (1982) System message design: Guidelines and experimental results. In: *Directions in Human/Computer Interaction* (A. Badre and B. Shneiderman, Eds.) Ablex, Norwood, NJ, pp. 55-78.
20. S. Ainsworth, P. Bibby, & D. Wood (2002) Examining the effects of different multiple representational systems in learning primary mathematics. *Journal of the Learning Sciences* **11,** 25-62.
21. T. Seufert (2003) Supporting cohrenece formation in learning from multiple represenations. *Learning and Instruction* **13,** 227-237.
22. D. Bodemer, R. Ploetzner, I. Feuerlein, & H. Spada (2004) The active integration of information during learning with dynamic and interactive visualizations. *Learning and Instruction* **14,** 325-341.
23. B. Shneiderman (1998) *Designing the User Interface: Strategies for Effective Human-Computer Interaction* Addison Wesley Longman, Menlo Park, CA.
24. M. Grossberg, R. Wiesen, & D. Yntema (1976) An experiment on problem solving with delayed computer responses. *IEEE Transactions on Systems, Man, and Cbernetics* **6,** 219-222.
25. G. L. Martin & K. G. Corl (1986) System response time effects on user productivity. *Behavior and Information Technology* **5,** 3-13.
26. R. Barber & H. Lucas (1983) System response time, operator productivity and job satisfaction. *Communications of the ACM* **26,** 972-986.
27. E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, & C. Cook (1997) Does continuous visual feedback aid debugging in direct-manipulation programming systems? In: *Proceedings CHI '97* ACM Press, New York, pp. 258-265.
28. C. Cook, M. Burnett, & D. Boom (1997) A bug's eye view of immediate visual feedback in direct manipulation programming systems. In: *Proceedings of the Seventh Workshop on Empirical Studies of Programmers* Erlbaum, Hillsdale, NJ.
29. A. T. Corbett & J. R. Anderson (1989) Feedback timing and student control in the LISP Intelligent Tutoring System. *Artificial Intelligence in Education***,** 64-72.
30. A. T. Corbett & J. R. Anderson (2001) Locus of feedback control in computer-based tutoring: impact on learning rate, achievement and attitudes. In: *Proceedings of the 2001 SIGCHI Conference on Human factors in Computing Systems* ACM Press, New York, pp. 245-252.
31. C. D. Hundhausen & J. L. Brown (2005) What You See Is What You Code: A radically dynamic algorithm visualization development model for novice learners. In: *Proceedings 2005 IEEE Symposium on Visual Languages and Human-Centric Computing* IEEE Computer Society, Los Alamitos, pp. 140-147.

32. P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, & P. Miller (1997) Mini-languages: A way to learn programming principles. *Education and Information Technologies* **2,** 65-83.

9. Tables

Table 1. Summary of the SALSA Pseudocode Language

COMMAND	MEANING
`create`	Creates new variables, arrays, and array indexes (special variables that reference array cells)
`set`	Creates and sets the value of new variables and array indexes on the fly, or changes the values of existing variables, arrays, and array indexes
`input`	Prompts the user for a variable’s value; the variable is created if it does not already exist
`populate`	Fills the empty cells of an array with new values
`print`	Outputs a value to the user
`if…elseif…else…endif`	Specifies blocks of code that execute conditionally based on the results of Boolean tests
`while…endwhile`	Specifies a loop that executes as long as the Boolean test evaluates to true
`move`	Moves a variable, array, or array index to a new location
`swap`	Causes two variables or array indexes to change positions

Table 2. Study tasks

TASK #	SYNOPSIS
1	Using ALVIS, design an algorithm that creates an array containing n random integers between 1 and 100. Your algorithm should compute the sum of all the values in the array, and print out that sum.
2	Using ALVIS, design an algorithm that creates an array containing n random integers between 0 and 2. Your algorithm should set all occurrences of the value 0 in the array to the value -1, and print out the number of such replacements that were made.
3	Using ALVIS, design an algorithm that first creates a "source" array containing n random integers between 1 and 100. Next, your algorithm should create a "destination" array of the same size containing all 0 values. Finally, your algorithm should copy the values of the "source" array to the "destination" array such that those values appear in the "destination" array in reverse order. .

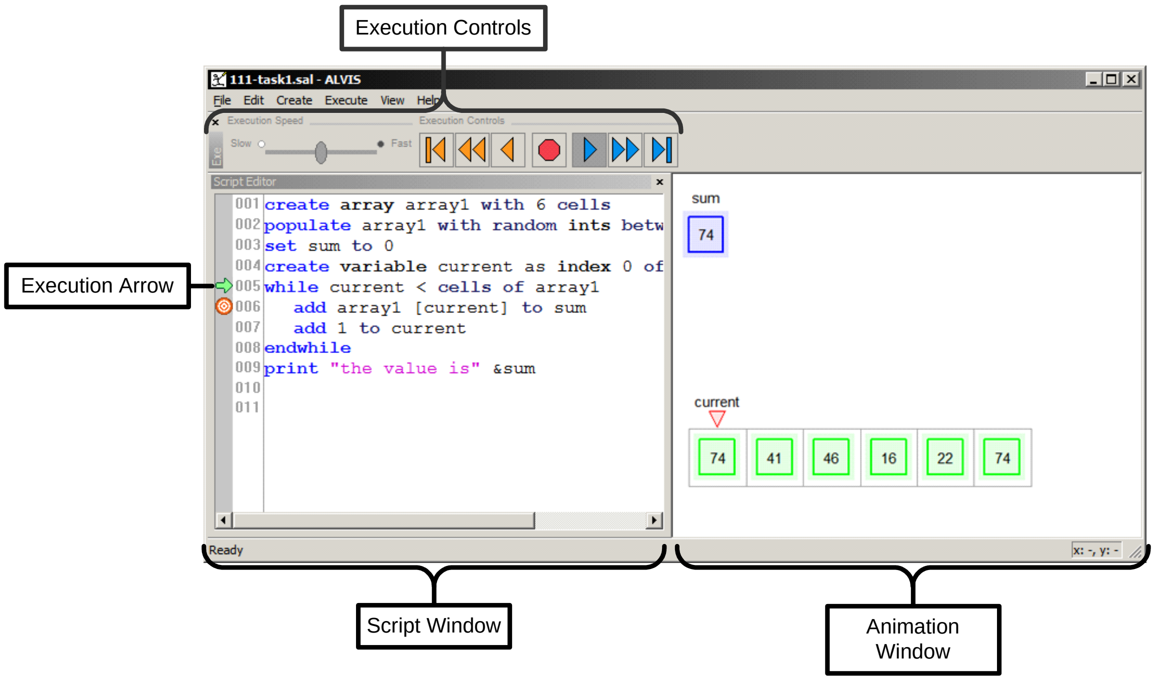
Table 3. Assignment of Participants to Treatment Orders

Treatment Order	*n*
No/Self-Select/Automatic (NSA)	10
No/Automatic/Self-Select NAS)	10
Self-Select/No/Automatic (SNA)	9
Self-Select/Automatic/No (SAN)	10
Automatic/No/Self-Select (ANS)	8
Automatic/Self-Select/No (ASN)	10

Table 4. Syntactic and Semantic Correctness by Treatment

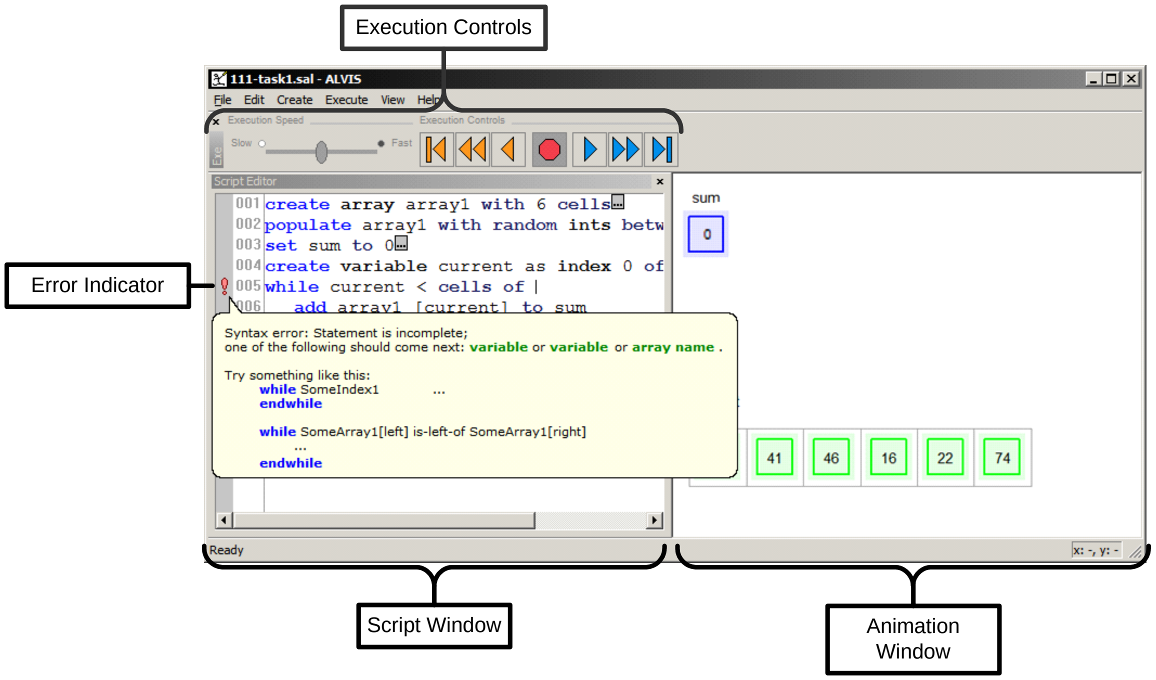
	Treatment		
Dependent Measure	Automatic	Self-Select	No Feedback
% Lines Syntactically Correct			
Min	50.00	54.55	33.33
Max	100.00	100.00	100.00
Mean	96.64	97.61	79.95
SD	10.13	7.86	19.34
% Semantic Components Correct			
Min	0.00	0.00	0.00
Max	100.00	100.00	100.00
Mean	69.55	74.55	62.23
SD	29.72	28.58	30.98

10. Figures



```
create array array1 with 6 cells
populate array1 with random ints betw
set sum to 0
create variable current as index 0 of
while current < cells of array1
   add array1 [current] to sum
   add 1 to current
endwhile
print "the value is" &sum
```

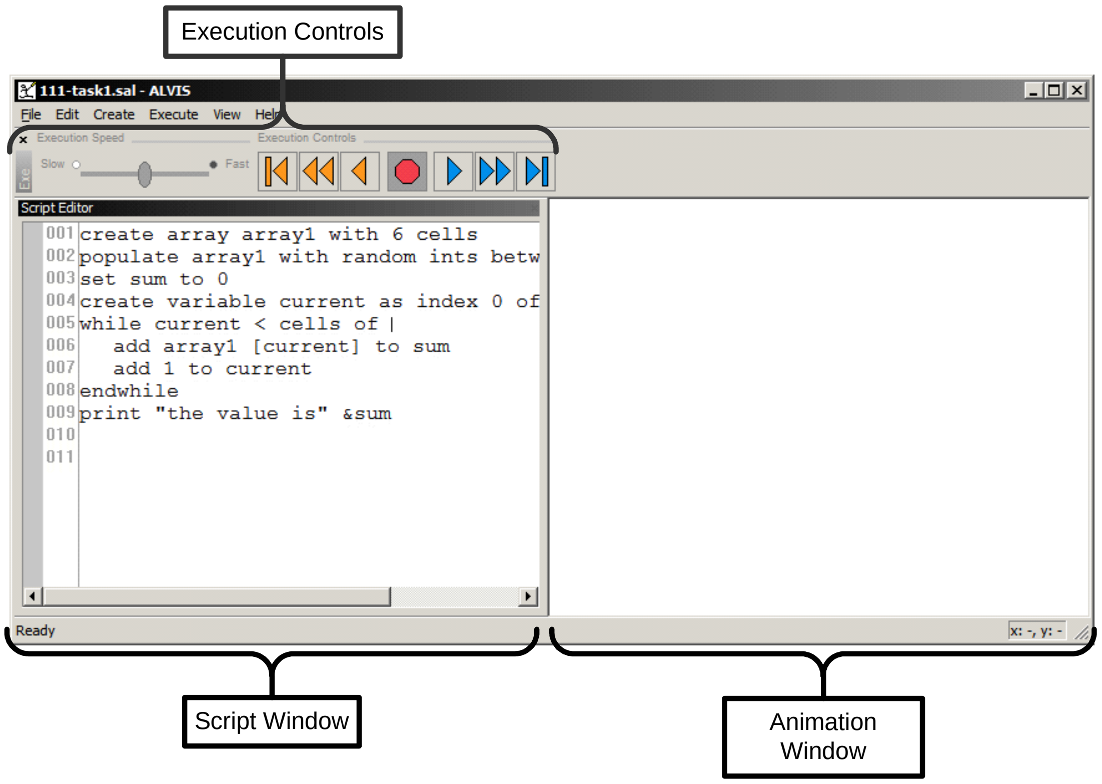
Figure 1. The programming environment used in the Automatic Treatment



```
create array array1 with 6 cells
populate array1 with random ints betw
set sum to 0
create variable current as index 0 of
while current < cells of |
   add array1 [current] to sum
```

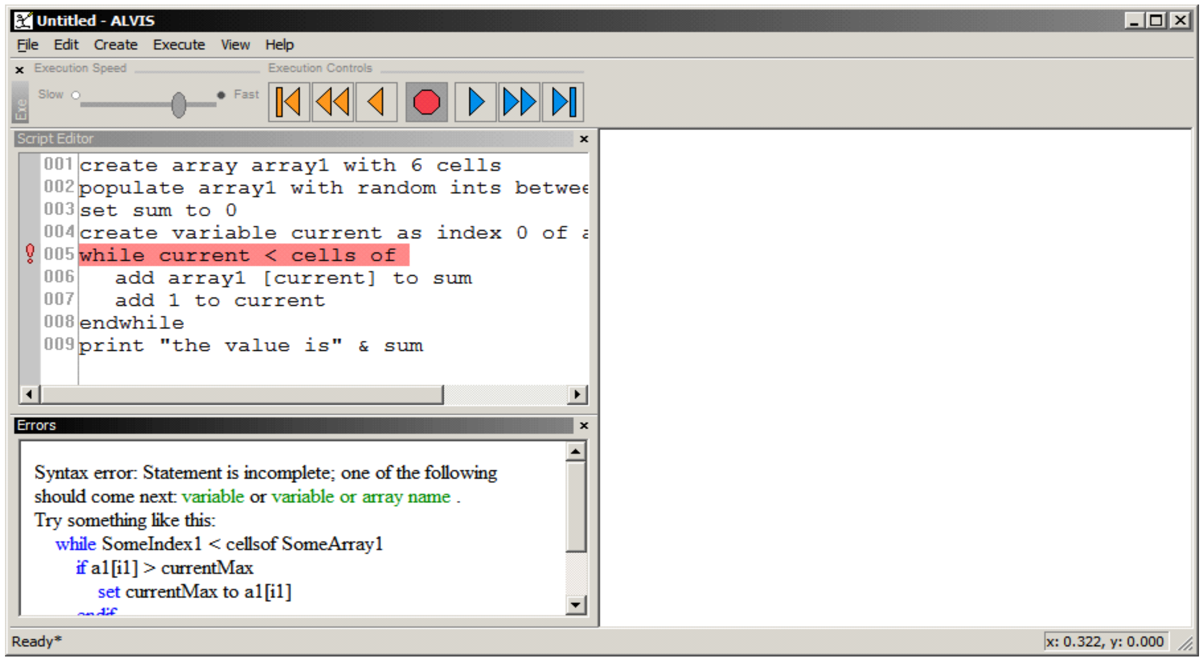
```
Syntax error: Statement is incomplete;
one of the following should come next: variable or variable or array name .

Try something like this:
     while SomeIndex1        ...
     endwhile

     while SomeArray1[left] is-left-of SomeArray1[right]
          ...
     endwhile
```

Figure 2. Up-to-the-keystroke syntactic feedback that appears in a pop-up tooltip in Automatic Treatment



```
create array array1 with 6 cells
populate array1 with random ints betw
set sum to 0
create variable current as index 0 of
while current < cells of |
   add array1 [current] to sum
   add 1 to current
endwhile
print "the value is" &sum
```

Figure 3. The programming environment used in the Self-Select treatment

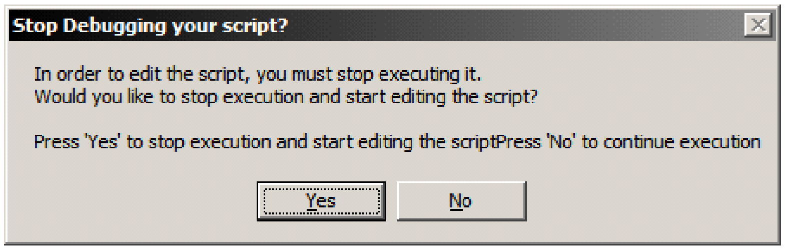


```
create array array1 with 6 cells
populate array1 with random ints betwee
set sum to 0
create variable current as index 0 of a
while current < cells of
   add array1 [current] to sum
   add 1 to current
endwhile
print "the value is" & sum
```

Syntax error: Statement is incomplete; one of the following should come next: variable or variable or array name .
Try something like this:

```
while SomeIndex1 < cellsof SomeArray1
   if a1[i1] > currentMax
      set currentMax to a1[i1]
```

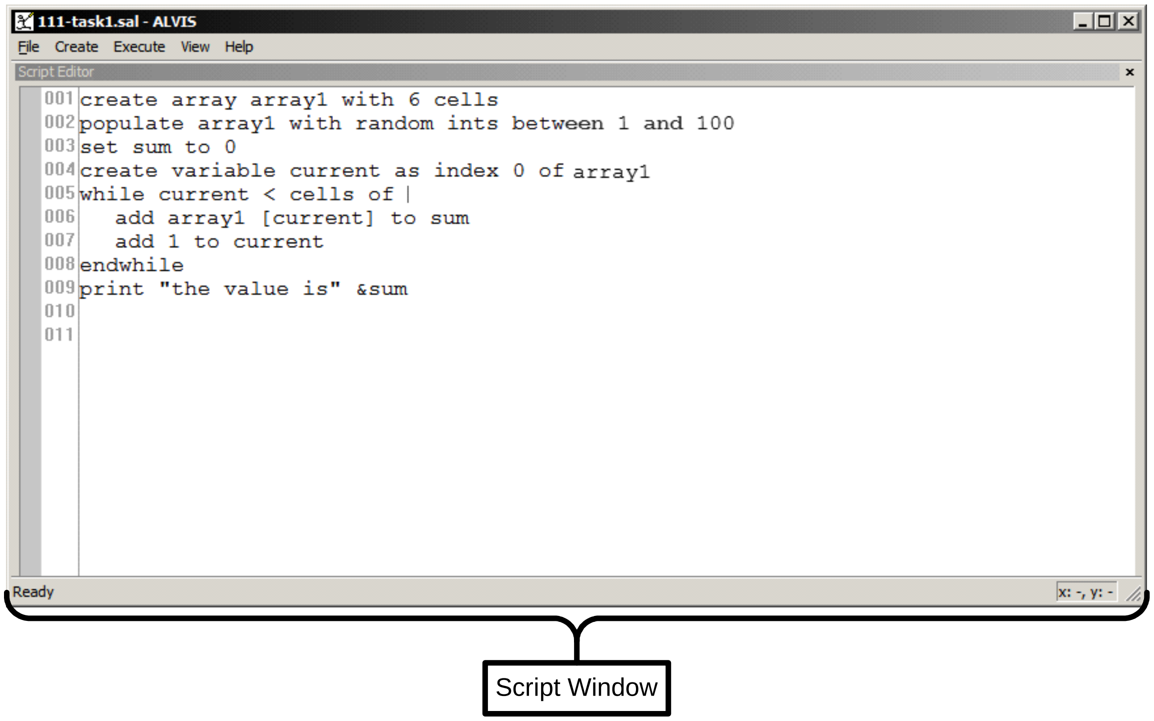
Figure 4. Example of how the Self-Select treatment software reports an error


Stop Debugging your script?

In order to edit the script, you must stop executing it.
Would you like to stop execution and start editing the script?

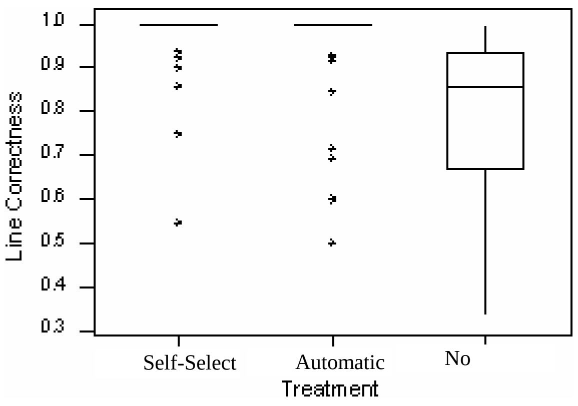
Press 'Yes' to stop execution and start editing the scriptPress 'No' to continue execution

Figure 5. Self-Select treatment dialog box that appears when the user attempts to edit a script when in "execution" mode

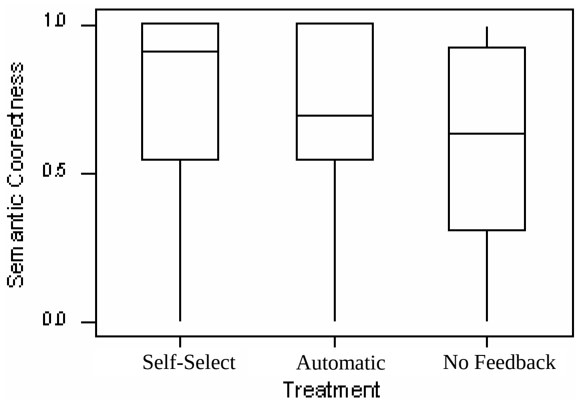


```
create array array1 with 6 cells
populate array1 with random ints between 1 and 100
set sum to 0
create variable current as index 0 of array1
while current < cells of |
   add array1 [current] to sum
   add 1 to current
endwhile
print "the value is" &sum
```

Figure 6. The programming environment used in the No Feedback Treatment

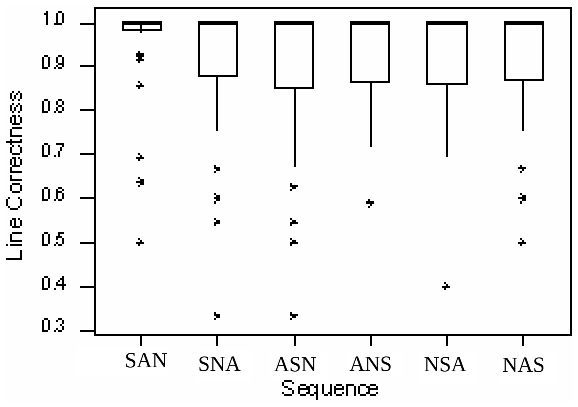


(a) % Lines Syntactically Correct

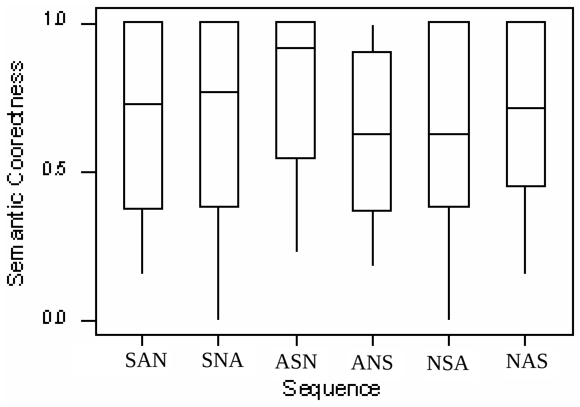


(b) % Correct Semantic Components

Figure 7. Box plots of (a) the syntactic (line) correctness and (b) semantic correctness by treatment. Note that the boxed regions delineate the middle 50% for each treatment; the asterisks denote outliers; and the horizontal lines through each box mark the medians.



(a) % Lines Syntactically Correct



(b) % Correct Semantic Components

Figure 8. Box plots of (a) syntactic (line) correctness and (b) semantic correctness by treatment sequence. Note that “S” is used to abbreviate the Self-Select treatment, “A” is used to abbreviate the Automatic treatment, and “N” is used to abbreviate the No Feedback treatment.