②

BRIDGE: INTELLIGENT TUTORING WITH INTERMEDIATE REPRESENTATIONS

Technical Report AIP - 21

Jeffrey G. Bonar & Robert Cunningham

Learning Research and Development Center
and Psychology Department
University of Pittsburgh
Pittsburgh, PA. 15260

The Artificial Intelligence and Psychology Project

Departments of
Computer Science and Psychology
Carnegie Mellon University

Learning Research and Development Center
University of Pittsburgh

2

BRIDGE: INTELLIGENT TUTORING WITH INTERMEDIATE REPRESENTATIONS

Technical Report AIP - 21

Jeffrey G. Bonar & Robert Cunningham

Learning Research and Development Center
and Psychology Department
University of Pittsburgh
Pittsburgh, PA. 15260

May 1988

This research was supported by the Computer Sciences Division, Office of Naval Research and DARPA under Contract Number N00014-86-K-0678, N00014-83-0148 and N00014-83-K06555; and by the Air Force Human Resources Laboratory under Contract Number F41689-84-D-0002, Order 0004. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not neccssarily reflect the views of the U. S. Government. Reproduction in whole or in part is permitted for purposes of the United States Government. Approved for public release; distribution unlimited.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AIP 21	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b. OFFICE SYMBOL *(If applicable)*	7a NAME OF MONITORING ORGANIZATION Computer Sciences Division Office of Naval Research
6c. ADDRESS *(City, State, and ZIP Code)* Department of Psychology Pittsburgh, Pennsylvania 15213	7b ADDRESS *(City, State, and ZIP Code)* 800 N. Quincy Street Arlington, Virginia 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Same as Monitoring Organization 8b OFFICE SYMBOL *(If applicable)*	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0678
8c. ADDRESS *(City, State, and ZIP Code)*	10 SOURCE OF FUNDING NUMBERS p4000ub201/7-4-86

PROGRAM ELEMENT NO	PROJECT NO	TASK NO.	WORK UNIT ACCESSION NO
N/A	N/A	N/A	N/A

11 TITLE *(Include Security Classification)*
Bridge: Intelligent Tutoring with Intermediate Representations (Unclassified)

12 PERSONAL AUTHOR(S)
Bonar, Jeffrey G. and Cunningham, Robert

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM 86Sept15 TO 91Sept14	14 DATE OF REPORT *(Year, Month, Day)* 1 May 1988	15. PAGE COUNT

16 SUPPLEMENTARY NOTATION
Proceedings of the Intelligent Tutoring Systems, June 1-3, Montreal

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Intelligent tutoring, Intermediate representations, Programming instruction, Computer Aided Instruction, Teaching Methods.

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

We describe an intelligent tutor called Bridge that provides support to novice programmers as they design, implement, and test programs. Not only does the tutor find and report student conceptual errors, but it also understands student designs and partially complete programs. This is done by providing intermediate representations that allow a student to directly represent designs and partial work. These intermediate representations are intended to give students specific mental models to support their problem solving process. Bridge supports a novice in an initial informal statement of a problem solution, subsequent refinement of that solution, and final implementation of the solution as programming language code. Students should finish the tutor with the ability to discuss their work at a conceptual level above that of their actual problem solution.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION	
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Alan L. Meyrowitz	22b TELEPHONE *(Include Area Code)* (202) 696-4302	22c OFFICE SYMBOL N00014

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Abstract – We describe an intelligent tutor called Bridge that provides support to novice programmers as they design, implement, and test programs. Not only does the tutor find and report student conceptual errors, but it also understand student designs and partially complete programs. This is done by providing intermediate representations that allow a student to directly represent designs and partial work. These intermediate representations are intended to give students specific mental models to support their problem solving process. Bridge supports a novice in an initial informal statement of a problem solution, subsequent refinement of that solution, and final implementation of the solution as programming language code. Students should finish the tutor with the ability to discuss their work at a conceptual level above that of their actual problem solution.

Introduction

Our goal is a programming environment that provides tutorial support to novice programmers as they design, implement, and test programs. Not only should the tutor find and report student conceptual errors, but it should also understand student designs and partially complete programs. Here, we report on the Bridge system, a step toward such an environment and tutor. In order to understand student designs and partial programs, Bridge provides languages that allow a student to talk about his or her designs and partial work. Providing such intermediate design languages has two implications for intelligent tutoring methodology. First, such languages enormously simplify diagnosis. Instead of deriving student intentions with elaborate partial matching based on a bug catalog – as used in the PROUST system described by Johnson [1986] – or a process model of the student's decision making – as done in the CMU Lisp Tutor [Reiser et al., 1985] – the tutor can ask a student about his or her intentions directly. Second, the use of intermediate languages gives students specific mental models with which to conceptualize their problem solving process. Students finish the tutor with the ability to discuss their work at a conceptual level above their actual problem solution.

The name Bridge comes from our intended "bridge" between novice and expert conceptions of programming. Novice conceptions of a problem solution are likely to be informal and sketchy. Bridge supports a novice in the initial informal statement of a problem solution, subsequent refinement of that solution, and final implementation of the solution as programming language code. This is accomplished in three phases:

- The first phase involves an informal statement and refinement of the goals for the program. The language we provide to the student is based on simple natural language phrases typically used when people write step-by-step instructions for other people. For example, a student can construct the phrase "keep doing these steps until the sum exceeds 100." The Bridge analysis of a student's phase I solution is based on a set of naive models of programming. Before moving to phase II, a student must develop a complete natural language problem solution.
- The second phase involves refining the informal description of phase I into a series of semi-formal programming plans. Plans are schema-like structures which describe how goals are transformed into actual programming code (see Soloway and Ehrlich [1985] for a complete description of plans). Plans typically have various roles that interrelate with the roles of other plans. Phase II of Bridge is designed to allow students to focus on relating various plan roles without the syntactic complexity required when relating a code representation of those roles. In

particular, phase II plans are atomic: all plan roles appear together with the plan. Phase II plan structures are runnable.

- The third phase requires translating the plan-based description of phase II into actual Pascal code. Students are provided with a Pascal structure editor [much like that of Garlan and Miller, 1984], and an interpreter with a stepping mode.

We begin with overview of the current Bridge implementation. Next, we discuss the decision to provide students with intermediate problem solving languages. We conclude with discussions of specific design decisions in Bridge and our experience with students using Bridge.

An Overview of the Bridge System

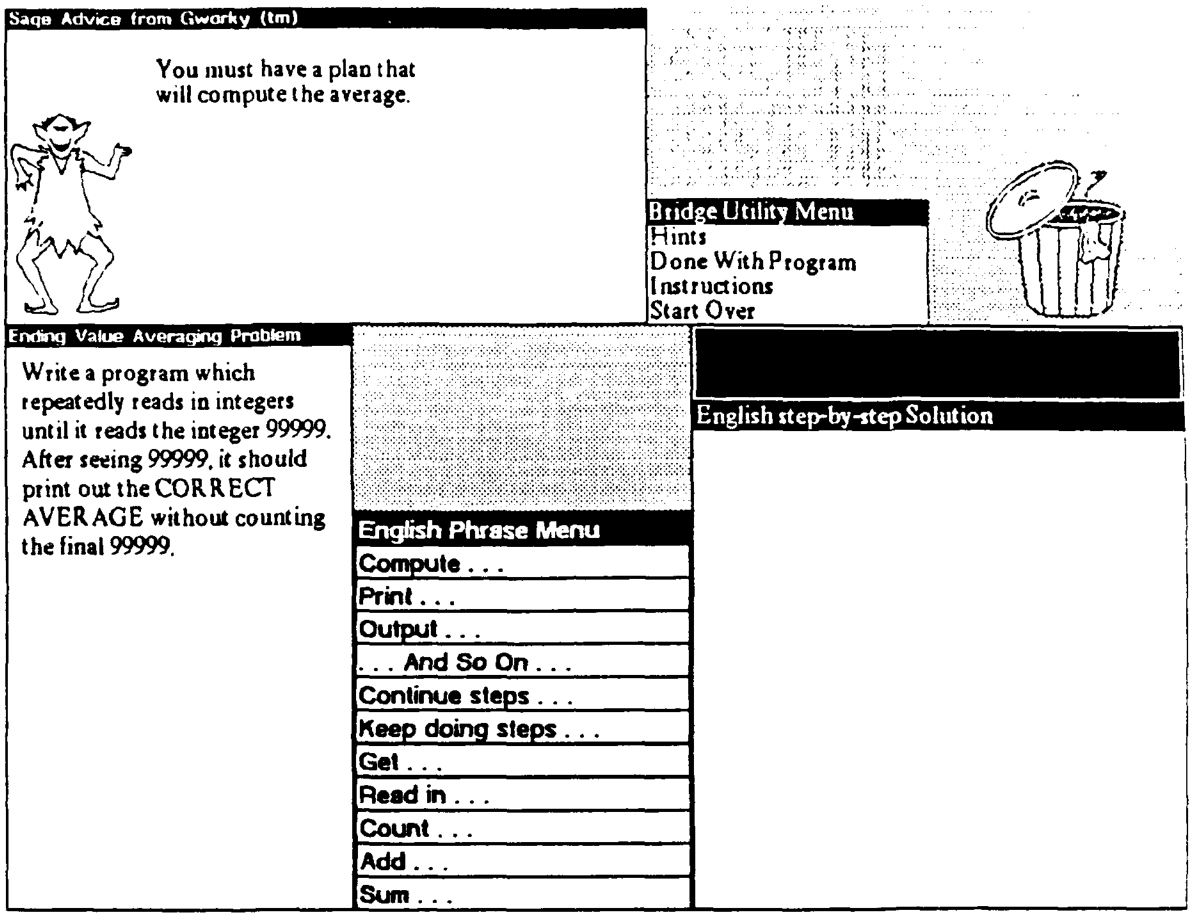
In Bridge, the student user is given a problem from the first ten weeks of a college level introductory programming course. The student passes through three phases while solving the problem.

In the first phase, the student constructs a set of step-by-step instructions using English phrases. In the next phase, the student matches these phrases to programming plans or plan fragments and builds a program using a representation of these plans. In the final phase, the student matches the plans to programming language constructs and uses these to build a programming language solution to the original problem. Currently the only language implemented in Bridge is Pascal, although many other programming languages could be used with the same approach.

We use an example problem to demonstrate the three phases of Bridge. The problem, called the "Ending Value Averaging Problem", is:

> Write a program which repeatedly reads in integers until it reads in the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999.

In the following description, several things should be kept in mind: (1) Bridge is highly interactive. The figures only give some sense of the display in action. Most of the objects of concern to the student are created, manipulated, and edited by moving screen objects with a mouse. (2) Bridge has been designed for students who have some familiarity with our particular version of programming plans. Since they are not covered in most texts, we have developed a programming plan workbook for our students (Bonar et al., 1986). We assume that users of Bridge are familiar with the plans in the workbook, up to the exercise they are attempting. Of course, students do not need to be skilled at using the plans, only familiar with the idea and basic purpose of plans. (3) The terminology and complex screen display of Bridge may seem overwhelming, particularly for a novice. The example problem presented is approximately one-half of the way through the programming curriculum we have designed. The problems students see initially are much simpler. The earliest problems consist of only one or two plans. A new problem never introduces more than one unfamiliar plan.



You must have a plan that will compute the average.

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999.

Figure 1. Bridge screen at the start of phase I.

Phase I: Building the Natural Language "Program"

Figure 1 shows the screen from Phase 1 with the ending value averaging problem as the current problem. The problem specification is in the lower left corner of the screen.

The user is to build his English language solution to the problem by making choices from the menu labeled "Natural Language Selections". The menu selections are the beginnings of phrases. Each phrase is intended to correspond to one plan. Some of these selections are redundant, for example, Print ... and Output .... This is done because our empirical work indicated that different people preferred different phrases to refer to the same plan (see Bonar [1985] and Miller [1981]).

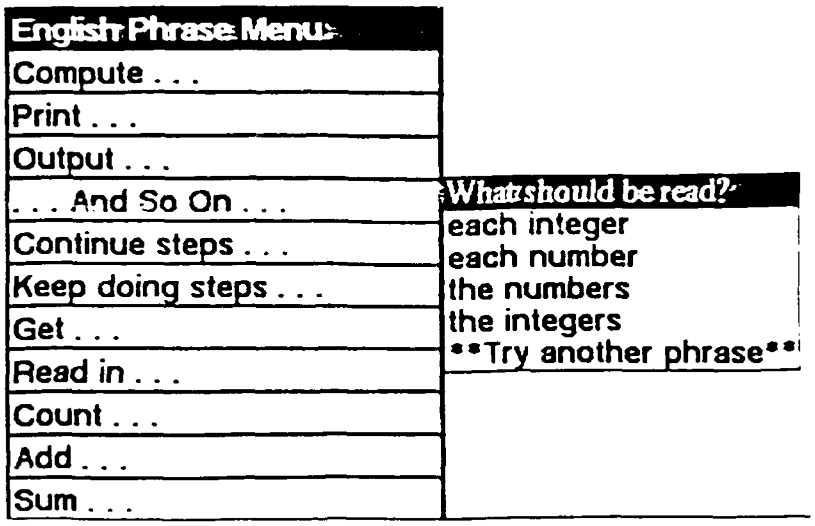
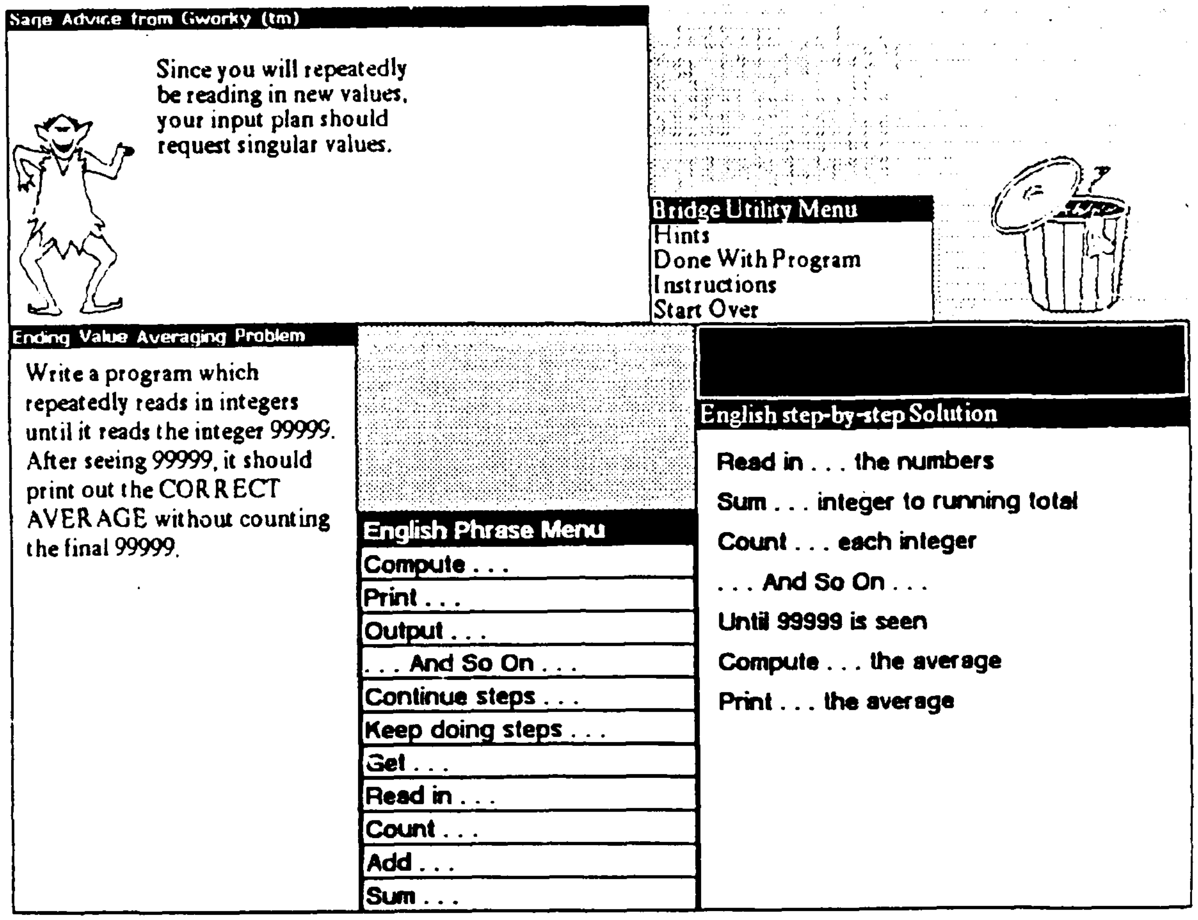


Figure 2. Phase I submenus allow the student to construct a specification that more closely represents his or her intentions. Here the submenu for the Read in phrase is shown.

After the student chooses the beginning of a phrase from the Natural Language Selections menu, a smaller submenu appears to the right of the selection. This submenu contains choices which complete the phrase (see Figure 2). In general, this second phrase further specifies or parameterizes the plan specified with the phrase from the Natural Language Selections menu. We allow a large amount of variability in surface lexicon with only a few basic underlying plans. In general, our goal is to allow the student to construct the English phrase that most precisely states his or her intentions, yet still map that phrasing into our catalog of programming plans.

Once a phrase is completed, it appears in the Natural Language Plans window. The student uses the mouse to move the phrase to a desired location and places the phrase with a button click. At each step, Bridge neatly formats the program for the student. The student builds the natural language program by combining phrases in an order that correctly specifies a solution to the problem. At any time during phase I the phrases may be repositioned or deleted from the program entirely. Figure 3 shows a typical phase I partial solution.


Sage Advice from Gworky (tm)

Since you will repeatedly be reading in new values, your input plan should request singular values.

Bridge Utility Menu
Hints
Done With Program
Instructions
Start Over

Ending Value Averaging Problem

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999.

English Phrase Menu
Compute . . .
Print . . .
Output . . .
. . . And So On . . .
Continue steps . . .
Keep doing steps . . .
Get . . .
Read in . . .
Count . . .
Add . . .
Sum . . .

English step-by-step Solution

Read in . . . the numbers
Sum . . . integer to running total
Count . . . each integer
. . . And So On . . .
Until 99999 is seen
Compute . . . the average
Print . . . the average

Figure 3. Typical phase I partial student solution.

Understanding a Student's Phase I Solution

If the student needs help or thinks the program is correct, he can select Hints or Done with program, respectively, from the main menu. In either case, Bridge checks the student program, offering appropriate tutorial advice or allowing the student to move to the next phase. The program responds to the student in the persona of Gworky the friendly troll. Gworky checks the program and coaches the student if there are any errors. To do this checking, the tutor builds a symbolic representation of the natural language version of the program. In this representation the tutor notes the order of each plan and the detailed phrasings used by the student off the submenus.

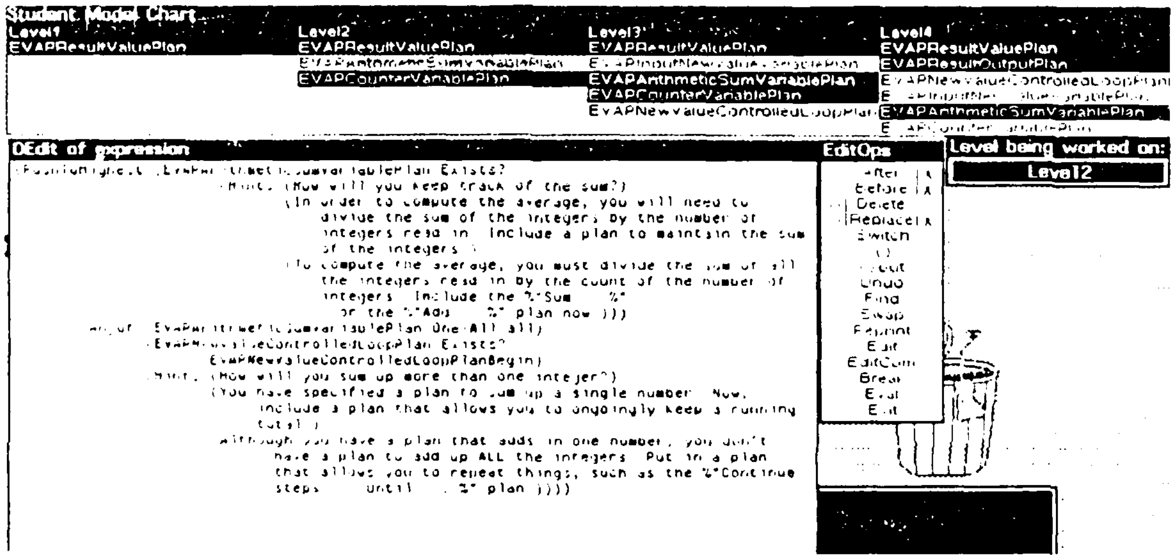
The symbolic representation of the natural language version of the program is compared with a list of *requirements* for a correct solution to the problem. The first requirement that the student fails to satisfy becomes the subject of Gworky's remarks. The requirements are supplied by the instructors as part of a problem description.

The requirements are specified at four different levels, corresponding to four different naive models of the programming process that a novice might bring to Bridge (see Biermann et al. [1983] and Bonar [1986]). These models differ from an expert model in that they do not specify all required plans, relax the constraints on slot fillers for the plans specified, and allow more flexible ordering than is required in an ideal solution. In essence, Bridge specifies four possible student models that can be matched against student phase I performance.

For example, a common naive model of looping allows a student to construct a loop with a description of the first iteration followed by the phrase "and so on." Based on the particular phrasings constructed by the student, Bridge infers a particular naive model. The specific coaching provided by Bridge is shaped by the naive model inferred and the student's current level of detail. Before moving to phase II, a student must develop a complete natural language problem solution using the most sophisticated (least naive) programming model.

The particular four levels are not ad hoc, but based on our empirical work with step-by-step natural language procedures. We have found that a novice's natural language solutions reflect one of four overall strategies for specifying looping in natural language [Bonar, 1986]. Since looping was the most complex aspect of the problems tutored in Bridge, we based our student model on the student's understanding of loops.

Figure 4 shows a schematic of the model with which the tutor diagnoses the student's solution. The four looping strategies (levels of the student model) are each captured by a set of requirements in a column of the chart shown in Figure 4. Each column shows the plans that must be present in the solution to satisfy that level. Each of those plans specifies a set of detailed requirements discussed in the next section.



Level1	Level2	Level3	Level4
EVAPResultValuePlan	EVAPResultValuePlan	EVAPResultValuePlan	EVAPResultValuePlan
	EVAPArithmeticSumVariablePlan	EVAPInputNewValueVariablePlan	EVAPResultOutputPlan
	EVAPCounterVariablePlan	EVAPArithmeticSumVariablePlan	EVAPNewValueControlledLoopPlan
		EVAPCounterVariablePlan	EVAPInputNewValueVariablePlan
		EVAPNewValueControlledLoopPlan	EVAPArithmeticSumVariablePlan
			EVAPCounterVariablePlan

```
(PositionRight (EVAPArithmeticSumVariablePlan Exists?
               (Hint: (How will you keep track of the sum?)
                      (In order to compute the average, you will need to
                          divide the sum of the integers by the number of
                          integers read in. Include a plan to maintain the sum
                          of the integers.)
                      (To compute the average, you must divide the sum of all
                          the integers read in by the count of the number of
                          integers. Include the %"Sum      %"
                            or the %"Add     %" plan now.)))
       Any of   EVAPArithmeticSumVariablePlan One All all)
          (EVAPNewValueControlledLoopPlan Exists?
                 EVAPNewValueControlledLoopPlanBegin)
          (Hint: (How will you sum up more than one integer?)
                 (You have specified a plan to sum up a single number. Now,
                      include a plan that allows you to ongoingly keep a running
                      total.)
                 (Although you have a plan that adds in one number, you don't
                      have a plan to add up ALL the integers. Put in a plan
                      that allows you to repeat things, such as the %"Continue
                      steps      until     . %" plan))))
```

Figure 4. A schematic of the model used to diagnose student solutions in phase I. Each column corresponds to a particular looping strategy and shows the plans that must be present in the solution to satisfy that level. Also shown is a detailed requirements specified by the Arithmetic Sum Variable plan.

The requirements have hints built in. If a specification is not satisfied, an associated hint is presented to the student. Most errors have three or more hints, increasing in specificity. Note that hints can be tailored to a particular unsatisfied requirement within a particular novice model.

When the student requests help by selecting Hints or Done with Program from a command menu at the top of the Bridge screen, the tutor then checks through the required plans, model by model, and matches the requirements from the plans at each model against the representation of the student solution. Once all plans are marked as satisfied or unsatisfied by the student's solution, a hint is selected to be presented to the student. In general, the student solution may fully satisfy the plans for one or more simpler looping strategies, and partially satisfy one or more sophisticated strategies. (That is, students don't work exclusively with one strategy at a time). We approximate the actual student understanding by selecting a hint from the simplest partially satisfied strategy.

In summary, the diagnostic strategy for Phase 1 is based on matching a student to a particular elementary model of programming. Each model requires particular plans expressed in informal natural language and a particular organization of those plans. When analyzing a student's informal specification of a solution, the system evaluates the student's satisfaction of the requirements for plans within each model. Tutoring (hint giving) is directed by the simplest unsatisfied plan in the simplest model with one or more unsatisfied plans.

Phase II: Building a "Plan Program"

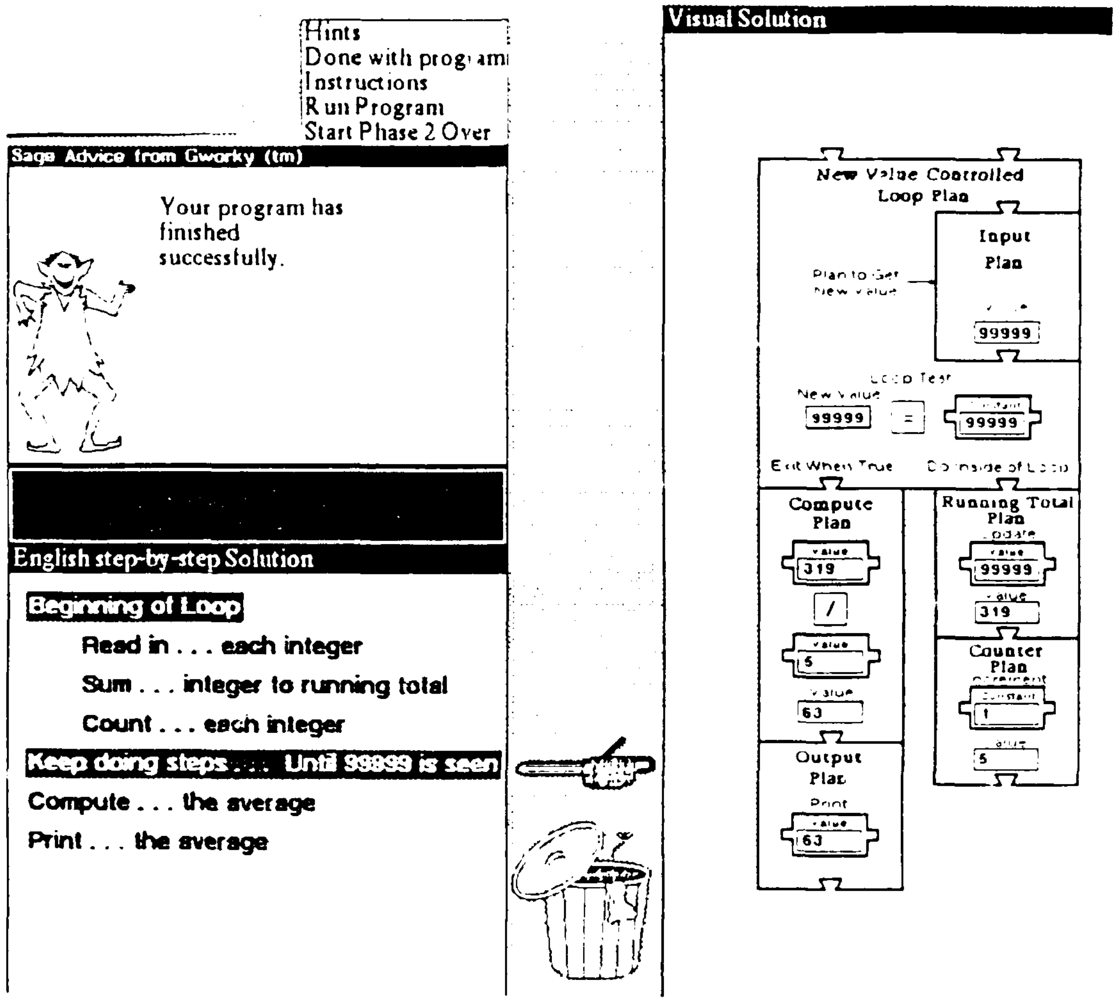
In phase II of Bridge a student takes the informal specification developed in phase 1 and formalizes the interconnection between the plans and their components. The student does this by selecting a line of the phase I specification and refining the intent of that line with one or more objects in a phase II plan specification language. Where in phase I the emphasis is on the appropriate solution strategy, accumulating the correct plan components, and ordering them in a reasonable way, in phase II the emphasis is on identifying the formal plan components and expressing their interrelationships.

Correct expression of plan interrelationships has been shown to be critically difficult for novices, particularly where those interrelationships result in the merging of several plans into one sequence of code [Mayer, 1979; Spohrer et al., 1985]. Phase II provides an environment where students can focus on plan interrelationships without the confounding concern of how those interrelationships will ultimately be expressed in code.

We have chosen to implement phase II as a visual programming language. Various constructs are implemented as icons that can be picked up, manipulated, placed, and connected together. Since the focus of phase II is on plans and their interconnections, we needed to develop a language that focused student attention on those elements. The essentially linear nature of textual languages made if very difficult to express interconnections. Furthermore, since plans represent rich, high-level programming objects, it is sensible to depict them as icons that suggest their function. Finally, the current iconic representation of phase II was suggested by comments from students confused while using earlier textual versions.

Design of the Bridge Phase II Visual Plan Language

The key design constraint of the phase II language is that each plan is represented by a single atomic icon. We feel this is crucial if we are to allow a student to focus directly on plan interconnection separate from other issues. The phase II plan icons used for the Ending Value Averaging Plan are shown in Figures 5. (A detailed discussion of the design of the phase II visual programming language is beyond the scope of this paper. See Bonar and Liffick [1988] for a complete discussion). The general metaphor of the plan language is that of puzzle pieces being fit together. Plans with similar shapes have similar roles in a program.



Your program has finished successfully.

Beginning of Loop
Read in . . . each integer
Sum . . . integer to running total
Count . . . each integer
Keep doing steps . . . Until 99999 is seen
Compute . . . the average
Print . . . the average

Figure 5. A phase II solution to the Ending Value Averaging Problem.

Plans that express a sequence of values in a variable, called *variable plans*, are shown as squares. Each of these plan icons embodies the entire semantics of a particular plan. The Counter plan, for example, keeps and shows the value of the count so far, knows to initialize itself to zero, and increments its value by a specified amount every time control flows through it. The Running Total plan similarly keeps and shows the value of the total so far, knows to initialize itself to zero, and adds the value of another plan into the total so far (how the connection with the plan that supplies that value is expressed is discussed below). The Input plan gets a sequence of values from the user of the program. Every time control passes through the plan a new value is requested from the user.

In addition to the variable plans, Figure 5 shows a Compute plan. The Compute plan uses values supplied by variable plans (tying a variable plan's value to other plans is described below) and an operation selected by the student. The operation is specified by mouse selection on the operator box and selecting an operator symbol off a pop-up menu. When control passes through the Compute plan the operation is performed with the current values of the associated variable plans.

The New Value Controlled Loop plan is one of the most complex plan icons in Bridge. There are four components of the New Value Controlled Loop plan: (1) *new value generator* – A variable plan to produce a series of values, each of which is tested to determine when to exit the loop. Typically the new value generator is an Input plan that requests values from the user. It could also be a random number generator or plan that traverses a data structure. (2) *end of loop condition* – A test to determine when to exit the loop. Note that this test is constrained to do a comparison to each new value. Another loop that requires a different sort of test – for example, to see if the running total has exceeded a particular value – requires a different plan. (3) *body of loop* – A series of plan icons to be executed in the body of the loop. (4) *actions to be performed after loop is complete* – A series of plan icons to be executed when the loop completes.

The key idea with the New Value Controlled Loop plan is to hide all the syntactic and control flow complexity that a student would need to confront to implement such a loop. The focus in phase II is to express the interconnection between the plans. How those interconnections are actually implemented in programming language code is the focus of phase III.

Figure 5 shows a phase II solution to the ending value averaging problem. The focus of phase II is on the connections between the plans. In particular, there are two kinds of connections that students must master: control flow and data flow. Control flow expresses the order of execution for the plans. In the phase II plan language, control flow is expressed explicitly by connecting the puzzle-piece tabs together. Plans are executed in a top-to-bottom order. So, for example, in Figure 5 the Compute Plan is executed before the Output Plan.

In addition to control flow, students must also express data flow. Specifically, students must show how values computed in one plan are used in other plans. This is done with a special plan called a Value plan. Value plans can be placed in holes within plans that are expecting values from another plan . Value plans are created by selecting the box labeled "Value" within a plan that can produce a value. Such a selection creates a Value plan icon that is attached to the mouse cursor. The data flow connection is established by dragging the Value plan to the plan that needs the value, and placing it in the appropriate hole.

Using Bridge Phase II

A student begins phase II with the phase I solution window (titled "Step-by-step English Solution") shifted to the left side of the screen and a new window titled "Visual Solution" is shown on the right side of the screen. Students work by selecting English phrases from the phase I solution. As each selection is made, the English phrase greys and the student is given the corresponding plan icon in the Visual Solution window. Icons can be moved about and placed as needed. As in phase I, students can request assistance from Gworky at any time. In addition, phase II solutions can be executed. Upon student request, a robust interpreter steps through the program, highlighting each plan as it executes, animating data flow with floating Value plans, and stopping to explain and suggest corrections for errors and omissions. A correct phase II solution requires all plans with all control and data flow connections correctly expressed.

Tutoring in phase II is very similar to tutoring in phase I, though less detailed. The student selects Hints whenever he or she needs help and Done with program when he or she thinks a correct plan solution to the problem has been formulated. The diagnosis is performed similarly to phase I, comparing the student's solution to the requirements.

Phase III: Building the Pascal Program

In phase III the student builds a programming language solution to the original problem. In particular, the goals and plans developed in phases I and II are finally realized as Pascal code. At the start of phase III the Visual Solution window from phase II is now shown on the left with a Pascal Solution window on the right. The student's task is to match each visual plan icon to one or more Pascal statements. Once statements are created, they are inserted and manipulated with a Pascal structure editor provided in the Pascal Solution window.

The student works by selecting a plan icon and then selecting a Pascal language construct from a pop-up menu of Pascal statements (see figure 6). If the student makes the correct choice, he or she is then asked to indicate the position for the statement in the Pascal Solution Window.

Bridge allows the student to avoid phase III syntax errors through a Pascal structure editor (see Garlen and Miller [1984] for a detailed discussion of structure editors). In addition to managing syntactic concerns, the editor also manages variable declarations.

Once the Pascal program is begun, the student may attempt to run it. Bridge works to run whatever pieces of program are in place. When execution reaches an unexpanded non-terminal node (from the structure editor) or an error, the program is halted with a detailed error message.

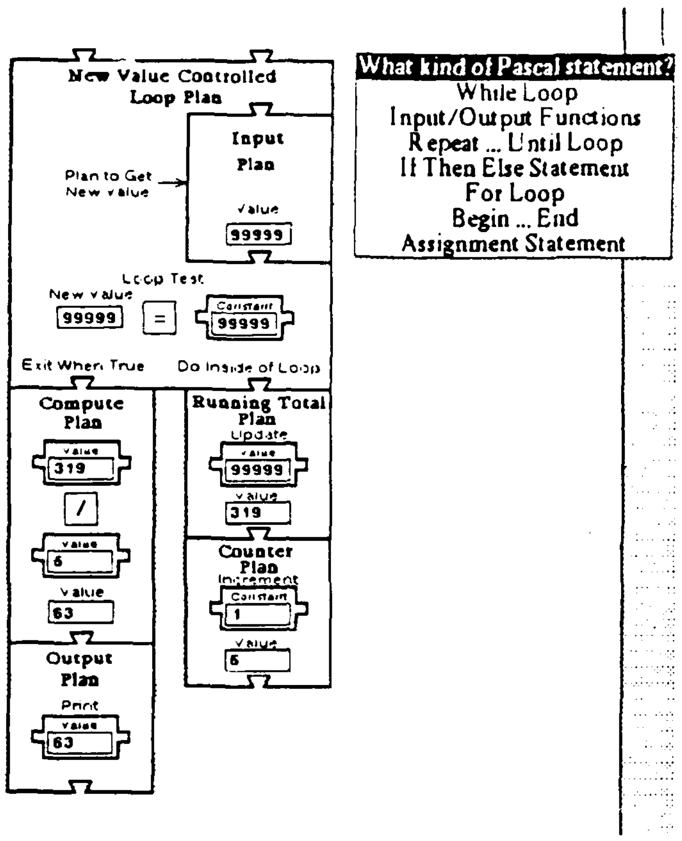


Figure 6. Selecting a Pascal construct in phase III of Bridge.

At the core of phase III is Bridge's supervision of the student matching process between phase II plans and Pascal constructs. This is not as regular a process as it might seem. In particular, one plan icon may require several Pascal constructs. When this occurs, Bridge tutors the student in moving from the plans to Pascal constructs. It is important to note that these complexities arise not from some peculiarity of Bridge but from the inherent irregularity of the mapping between Pascal and the underlying goal structure of Pascal programs. In fact, the main purpose of phase III is to support the student in learning about moving from an explicit goal structure to the peculiar realization of that goal structure in Pascal.

Intermediate Languages For Problem Solving

The Bridge approach gives the student languages for explicit manipulation of designs and partially complete programs. The approach is motivated by two complementary concerns. The first concern relates to extending the capability and ease of specification for the part of Bridge that understands student errors – the diagnostic component. Our second concern relates to what is known about novices learning to program. In this section we discuss these two issues.

Simplifying Diagnosis with Intermediate Languages

In any intelligent tutoring system a key problem is inferring student intentions from student behavior. In particular, the tutor must infer all mental activity from the final solution presented by the student. In the programming domain, for example, a standard intelligent tutor must reconstruct the student's entire mental activity between seeing a program specification and actually entering code in the machine. Such a reconstruction must account for both the correct and incorrect knowledge used by the student during design and implementation.

With a knowledge base of correct and buggy operators a tutor can use search techniques to reconstruct plausible accounts for errorful student responses to problems. This approach has been powerfully demonstrated in the programming tutor PROUST [Johnson, 1986]. While the approach works, it is very costly in terms of both search time and knowledge engineering. The accomplishments of PROUST must be weighed against the large cost in knowledge engineering time – several hundred hours per problem tutored [Johnson, 1986b]. This knowledge engineering is particularly cost ineffective because the student sees so little of the results. That is, almost all of the knowledge engineering that has gone into representing correct and buggy student actions is never seen by the student. Inside of PROUST, these operators are optimized for the the search task. They are not available in a form that could be presented to students, or used to assist students as they work toward a solution.

Bridge uses a different approach to reconstructing the student's intentions. Instead of attempting to reconstruct a student's entire reasoning from problem statement to final code in one step, Bridge has the student prepare intermediate solutions in languages that correspond to particular levels in the goal decomposition and the goal-to-code translation processes. This alleviates many of the difficulties of the PROUST approach. The search is more manageable because it has been broken up into a series of much smaller searches. In each of the smaller searches there are fewer relevant operators to try and less reasoning "distance" to span between the student's surface behavior and the solution the tutor is trying to reconstruct.

How is it that the relatively simple requirements matching process can give sufficient diagnostic power to deal with most student errors? We have the crucial advantage that we understand the student's intentions, based on their phase I and II solutions. Each piece of the early solutions is tied to a statement of intentions. Instead of recreating the entire goal-plan problem solving trace, as done in PROUST [Johnson, 1986] or the CMU Lisp tutor [Reiser et al., 1985], we are tracking the trace a level at a time, focusing on the critical difficulties of that level. Phase I corresponds to the highest level expression of the goals. Phase II allows the student to refine those goals and begin to add implementation information. In phase III the refined goals are actually turned into Pascal code.

In addition, the Bridge approach simplifies the knowledge engineering. The fewer operators in each search correspond to a smaller catalog that must be loaded into the program. In addition, the knowledge engineer has an easier task because the operators are simpler and it is easier to tell when the space of possible correct and errorful versions has been covered.

Pedagogical Arguments for Intermediate Languages

The Bridge approach grew from our concern with the conceptual distance between the syntax and semantics of programming languages like Pascal and the purpose and goals realized by that code. We felt that the programming task, as typically presented, confronts students with an enormous gap between goals and final product (e.g. Pascal code). The intermediate languages in Bridge are intended to "bridge" this gap.

To illustrate the gap between goals and code in Pascal, consider the goal of "keeping a count" as implemented in Pascal: (1) above the loop – a programmer must declare a counter variable and initialize it to zero using an assignment statement; (2) inside the loop – the counter must be incremented, again using an assignment statement with a peculiarly non-algebraic construction: `Count := Count + 1`; (3) below the loop – the counter value must actually be used below the loop. In summary, the simple goal of "keeping a count" has been spread throughout the program and buried in general purpose constructs. The design of these constructs is more closely related to the architecture of a register-based computer than to the problem actually being solved in the code.

Research into how novices learn programming reveals that the semantics of typical programming languages are not closely related to the way a typical novice understands a program [Bonar and Liffick, 1988]. Success with programming seems to be tied to a novice's ability to recognize general goals in the description of a task and to translate those goals into the relatively foreign program code (see, for example, Anderson and Jeffries [1985], Eisenstadt et al. [1981], Mayer [1979], or Soloway and Ehrlich [1985]). The Bridge approach allows students to explicitly represent their goals, describe how those goals interrelate, and only then concern themselves with how to translate those goals into code. The different tasks in programming are separated, simplifying the learning task.

In general, the Bridge approach can be seen as a collaboration between the student and the system (see Collins et al. [1987]). As discussed above, the value of such an interactive relationship lies in Bridge teaching expert strategies in a problem solving context shared with the student. Bridge also provides scaffolding for the selection and application of those strategies.

Principled Design Decisions in Bridge

The key principle of Bridge is:

> Teaching elementary programming with intermediate representations provides novice programmers with a deeper understanding of the programming language and process then is possible with conventional approaches.

The students learn programming, we assert, because they are facilitated in developing the mental models necessary to successfully carry out each phase of programming. So, then, the decision to present Bridge students with a series of phases, each built around a particular representation form, is principled.

Furthermore, the particular phases chosen are principled. The phases have developed out of a large body of research into the mental representations of novice and expert programmers. Each phase represents a particular set of concerns and programming sub-skills.

In phase I we ask students to state the key components of the problem solution, order those components correctly, and mention those components consistent with a "programming like" (as explained above) mental model of program loops. The language used in phase I is natural language via menu selections. The particular phrases used in the menus come from the phrasings used by subjects in a study of non-programmers writing step-by-step natural language instructions [Bonar, 1985].

In phase II we ask students to transform their phase I solution into a plan-like solution. The plans chosen are based on classroom experience with a plan based curriculum [Bonar et al., 1986]. In phase II students must confront the full complexity of programming control structures and data flow but without the added complexity of programming language syntax. Combining the plans with concern for programming language control and data flow are central to a beginning programmer's understanding (see, for example, Soloway and Ehrlich [1985]).

In phase III we ask students to transform their phase II plan structures into a Pascal solution. In this phase students must take a fairly complete solution and map it into the particular syntactic features of Pascal. In terms of Shneiderman and Mayer's [1979] syntactic/semantic theory, they must use their programming language specific knowledge of Pascal.

Using Bridge with Students

We have run Bridge with approximately 40 students ranging from no programming experience to half way through a college level introductory Pascal programming course. Bridge currently contains 25 problems. Each problem took approximately one person-week to be specified and fully described to Bridge.

While there is no systematic evaluation, there are some highlights from our experience with Bridge. First, students who are near the beginning of their programming course or doing poorly in that course appreciate phase I much more than students near the end of a course or doing well in a course. This is as expected: one would expect that more advanced students have less need for the informal representation allowed in phase I. At one point in our testing we ran Bridge with three students doing very poorly in their current programming course. In each case the use of Bridge was found by the student and his or her instructor to be very helpful. This suggests that Bridge may be particularly helpful for beginning programmers.

The riskiest part of the Bridge design is phase II. Despite the potential problems with phase II, we find it to be used successfully by both good and poor students. In particular, students reported that the use of phase II clarified their understanding of the problem and the programming constructs. It is seen by students as a place they can work on their programs without worrying about the details of Pascal. Students spend the majority of their time working in phases I and II of Bridge. Phase III is seen by the students as a way to produce the final product, not as a way to think about and plan the program.

We plan to use Bridge with 10-15 problems in a introductory Pascal class in Fall of 1988. At that time we will complete a detailed evaluation of Bridge and our overall approach.

Concluding Remarks

The work reported here focuses on the explicit use of intermediate representations to teach introductory programming. We have sought, through these intermediate representations, to provide students with an environment where they can talk about and receive tutoring about their designs and partial work. Specifically, Bridge supports a novice in the initial informal statement of a problem solution, subsequent refinement of that solution, and final implementation of the solution as programming language code. Bridge provides explicit coaching for a student in working through parts of the programming process that are usually implicit. The power of the computer's dynamic and graphical capabilities, coupled with the immediacy of intelligent tutoring have provided us with the capability to coach a student in a detailed and conceptually rich way.

Our goal with Bridge is not that students merely learn the constructs of a programming language, but that students gain the ability to discuss their work at a conceptual level above that of their actual problem solution. Although our success with this goal is not yet fully tested, initial results are very encouraging. We are eager for the results of our detailed study.

Bridge invites a more radical possibility than more effective tutoring of traditional programming languages. Except for syntactic and idiosyncratic semantic details, the program is fully specified at the end of phase II. Why should the student, or the typical programmer, be forced to learn and use those details? We have been actively investigating the notion of an intention-based programming language with constructs at the level of plans (see Bonar and Liffick [1988]).

We are interested in developing principles for use of intermediate representations, like those in phases I and II, for domains other than programming. What if, for example, students could use similar structures to reason about mechanics problems, geometry proofs, algebra word problems, or even arguments in an essay? The possibility is intriguing for two reasons. Plan-like structures have played a central role in a cognitive understanding of learning and thinking (see, for example, Resnick, 1983). A seemingly important gap in research with these structures exists in that they have not been used directly for instruction. A Bridge-like framework allows us to explore this possibility.

Acknowledgments

This work was supported by the Air Force Human Resources Laboratory under contract number F41689-84-D-0002, Order 0004 and by the Office of Naval Research under contract numbers N00014-83-6-0148 and N00014-83-K0655. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

The earliest version of the Bridge system was programmed by Mary Ann Quayle. Paul Beatty, John Corbett, Vikki Pitts, Jamie Schultz, and Bill Weil contributed substantially to the development of the current version of Bridge. We are grateful to the Intelligent Tutoring

Systems group for their suggestions and encouragement in this project. Many useful comments on earlier drafts were provided by John Seely Brown, Robert Glaser, Alan Lesgold, Stellan Ohlsson, Lauren Resnick, and Robert Rist.

References

Anderson, J. R. and Jeffries, R. [1985]. Novice LISP Errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1, 107-131.

Biermann, A.W., Ballard, B.W., Sigmon, A.H. [1983]. An Experimental Study of Natural Language Programming. *International Journal of Man-Machine Studies* 18:71-87, 1983.

Bonar, J. [1985]. Understanding the bugs of novice programmers. Unpublished doctoral dissertation. University of Massachusetts, Amherst.

Bonar, J. [1986]. Mental Models of Programming Loops. Technical Report, Intelligent Tutoring Systems Group, Learning Research and Development Center, University of Pittsburgh, PA 15260.

Bonar, J. & Liffick, B. [1988]. A Visual Programming Language for Novices. In press, to appear in *Visual Languages and Visual Programming*, edited by Shi-Kuo Chang, Prentice Hall Publishers.

Bonar, J., Weil, W., & Jones, R. [1986]. The Programming Plans Workbook. Technical Report, Intelligent Tutoring Systems Group, Learning Research and Development Center, University of Pittsburgh, PA 15260.

Bonar, J., & Soloway, E. [1985]. Pre-programming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1, 133-161.

Collins, A., Brown, J.S., and Newman, S.E. [1987]. Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. In *Cognition and Instruction: Issues and Agendas*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Eisenstadt, M., Laubsch, J., and Kahney, H. [1981]. Creating Pleasant Programming Environments for Cognitive Science Students. In Proceedings of the Third Annual Cognitive Science Society Conference. Cognitive Science Society, Berkeley, California.

Garlen, D.B. & Miller, P.L. [1984]. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. In Proceedings of the Software Engineering Symposium on Practical Software Development Environments. ACM-SIGSOFT/SIGPLAN, April.

Johnson, W. L. [1986]. *Intention-Based Diagnosis of Errors in Novice Programs.* Morgan Kaufman, Palo Alto, CA.

Johnson, W. L. [1986b]. personal communication.

Mayer, R. E. [1979]. A Psychology of Learning BASIC. Communications of the Association For Computing Machinery Vol. 22, No. 11, pp. 589-59.

Miller, L. A. [1981]. Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Systems Journal*, 20, pp. 184-215.

Reiser, B., Anderson, J., & Farrell, R. [1985]. Dynamic Student Modelling In an Intelligent Tutor For Lisp Programming. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pp. 8-14.

Resnick, L. [1983]. A New Conception of Mathematics and Science Learning. *Science*, Vol. 220, pp. 477-478.

Schneiderman and Mayer [1979]. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences*, Vol. 10, No. 3, pp. 219-238.

Soloway, E., & Ehrlich, K. [1985]. Empirical studies of programming knowledge. *IEEE Transactions of Software Engineering*, SE-10, November, pp. 595-609.

Spohrer, J., Soloway, E., & Pope, E. [1985]. A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, 1, pp. 163-207.