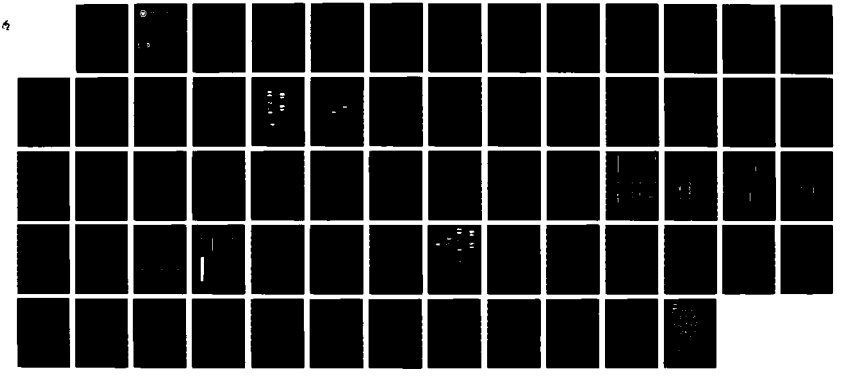
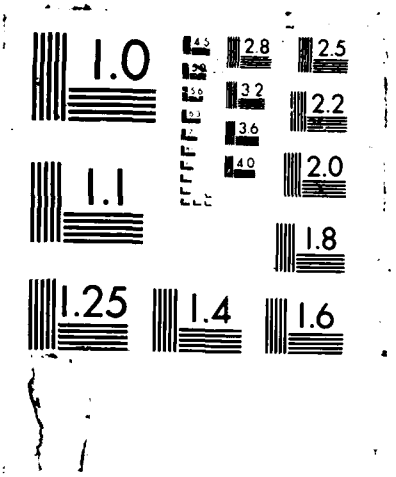
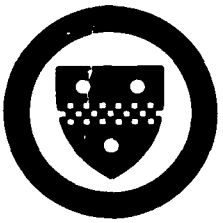
AD-A189 420 A VISUAL PROGRAMMING LANGUAGE FOR NOVICES(U) PITTSBURGH 1/1
UNIV PA LEARNING RESEARCH AND DEVELOPMENT CENTER
J G BONAR ET AL. 30 SEP 87 UPITT/LRDC/ONR/LSP-5
UNCLASSIFIED N00014-86-K-0678 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

4



University of Pittsburgh

DTIC FILE COPY

A Visual Programming Language for Novices

AD-A189 420

Jeffrey G. Bonar

Learning Research and Development Center
University of Pittsburgh

Blaise W. Liffick

Department of Mathematics and Computer Science
Millersville University

DTIC
ELECTE
FEB 1 8 1988
S D

September 30, 1987

Technical Report No. LSP-5

This research was sponsored by the Office of Naval Research, University Research Initiative, Contract No. N00014-86-K-0678.

Reproduction in whole or part is permitted for any purpose of the United States Government.

Approved for public release; distribution unlimited.

88 2 17 002

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION	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): UPITT/LRDC/ONR/LSP-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Learning Research & Development Center, University of Pittsburgh; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Cognitive Science Program, Office of Naval Research (code 1142CS)
6c. ADDRESS (City, State, and ZIP Code): 3939 O'Hara Street, Pittsburgh, PA 15260	7b. ADDRESS (City, State, and ZIP Code): 800 North Quincy Street, Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING 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: PROGRAM ELEMENT NO 61153N; PROJECT NO RR04206; TASK NO RR04206-00; WORK UNIT ACCESSION NO NR442c524

11. TITLE (Include Security Classification)

A VISUAL PROGRAMMING LANGUAGE FOR NOVICES (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)

Jeffrey G. Bonar and Blaise W. Liffick

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1987, September 30	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
05	08		Visual languages; novice programming; programming education; programming cognition
23	02		

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

We present BridgeTalk, a new approach to visual languages for novice programmers. The design of BridgeTalk is based on data about how novices learn to program. BridgeTalk allows novices to program with programming plans -- frame-like objects that capture essential program components like "keep a running total" and "iterate down a data structure". Novices are focused on the interactions between plans, not on the implementation details for a particular plan. Beginning with plans as a basis for a novice programming language, we were forced to develop a programming formalism that can deal with multiple levels of detail, merged plan implementations, and interrelationships between plans. The actual visual presentation for the language is based on six "design, implement, test with students, and redesign" cycles.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Susan M. Chipman	22b. TELEPHONE (Include Area Code): 202-696-4318; 22c. OFFICE SYMBOL: ONR 1142pt

DD Form 1473, JUN 86 — Previous editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

A Visual Programming Language for Novices

Jeffrey G. Bonar
Computer Science Department and
Learning Research and Development Center
University of Pittsburgh

Blaise W. Liffick
Department of Mathematics and Computer Science
Millersville University

September 1987

Abstract

We present BridgeTalk, a new approach to visual languages for novice programmers. The design of BridgeTalk is based on data about how novices learn to program. BridgeTalk allows novices to program with programming plans — frame-like objects that capture essential program components like "keep a running total" and "iterate down a data structure ". Novices are focused on the interactions between plans, not on the implementation details for a particular plan. Beginning with plans as a basis for a novice programming language, we were forced to develop a programming formalism that can deal with multiple levels of detail, merged plan implementations, and interrelationships between plans. The actual visual presentation for the language is based on six "design, implement, test with students, and redesign" cycles.

To Appear in *Visual Languages and Visual Programming* edited by Shi-Kuo Chang, Plenum Publishing Company, 1988.

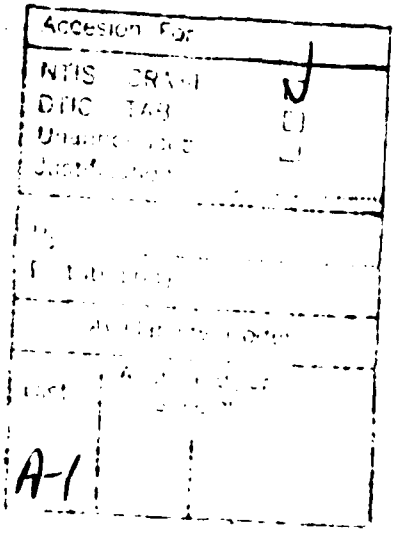
Comments To: Jeffrey G. Bonar
Intelligent Tutoring Systems
Learning Research and Development Center
University of Pittsburgh, PA 15260
412/624-7047
bonar@pitt.csnet

Acknowledgements

BridgeTalk came out of a richly interdiciplinary research community of computer scientists, psychologists, programmers, and educators, all concerned with improving the quality and impact of educational applications of computers. Discussions with Alan Lesgold and Lauren Resnick helped to crystalize the ideas presented here. Mary Ann Quayle, Jamie Schultz, and John Corbett contributed to the design and implementation of early versions of BridgeTalk. Bob Cunningham contributed substantially to the design of later versions and was the principle programmer on the project. Paul Beatty, Vikki Pitts, and Perry Riggs also contributed to later versions. The Chips interface design tool was developed and implemented by John Corbett and Bob Cunningham. An earlier version of the discussion on novice programming language design appeared in *Technology and Learning*, published by Lawrence Erlbaum Associates.

The work reported here was supported by the Office of Naval Research under their University Research Initiative, contract number N00014-86-K-0678. The Learning Research and Development Center AI Equipment and software base was developed under Office of Naval Research contract numbers N00014-83-6-0148 and N00014-83-K0655. The original version of Bridge was developed under Air Force Human Resources Laboratory contract F41689-84-D-0002, order 0004. The Faculty Academic Development Committee of Millersville University provided support for Mr. Liffick's work.

The opinions expressed do not necessarily reflect the position or policy of the agencies and no official endorsement should be inferred.



"But if we look at what a programmer would say about a program to a colleague who wanted to work on it or use it, very little of the description appears anywhere in the code."

Winograd [1979]

1. Introduction

There is a wealth of evidence that experienced programmers use a large set of programming plans: standard templates and structures for accomplishing typical programming tasks [Bonar and Soloway, 1985; Soloway and Ehrlich, 1984; Spohrer, 1985; Waters, 1985]. Example plans include "running total" and "iterate through a data structure looking for a distinguished value". Our objective is to take advantage of plans to provide novice programmers with an environment for learning to program. This environment begins with a high-level, plan based language, but permits and encourages the ultimate use of a standard language like Pascal. For several reasons the plan language has been designed as a visual language. This paper reports our initial efforts at formalizing and generalizing this plan language.

In learning a programming language, novices have two fundamental difficulties, both of which are addressed in our plan language:

> *Relating experience with informal plans to programming:* Empirical evidence [Bonar and Soloway, 1985; Kahney and Eisenstadt, 1982] suggests that novice programmers bring a vocabulary of programming-like plans from everyday experience with procedural specifications of activities expressed in natural language. These plans come from experience with step-by-step instructions like "check all the student scores and give me an average" or "see that hallway, if any doors are open close them." These informal plans, however, are often extremely difficult for novices to reconcile with the much more formal plans used in standard programming languages. Note, for example, that both example phrases involve an iteration without any specific mention of a repeated action.
>
> *Translating formal plans into programming constructs:* Even if a student recognizes particular formal plans, they are likely to have difficulty translating plans into programming code. A running total, for example, is implemented in Pascal with four statements, dispersed throughout a program: a variable declaration, an initialization above a loop, an update inside that loop, and a use below the loop. Spohrer et al. [1985] have shown that plan-to-code translation errors account for many student errors.

In responding to these problems, we have developed four objectives for our visual plan-based programming language for novices:

(1) Allow users to "connect" to their informal (primitive) plans. Unless novices can recognize how their own understanding of plans fits into the programming environment they are being faced with, they will find it impossible to formulate correct solutions.

(2) Support novice programmers in learning a "vocabulary" of programming plans. Eventually, the user would begin to think in terms of the programming plans themselves, not the informal plans. Not only is there a catalog of plans, but students are able to create their own plans.

(3) Support novice programmers in learning how to implement plans with a standard programming language. The ultimate goal is that the user will learn enough to be able to program in a standard programming language such as Pascal.

(4) Support the use of plan-like composition of programs. Plans can be seen as the essence of good program comments. To the extent that a program uses a vocabulary of plans, the program will be easy to read and understand.

These goals create an environment that supports a new, higher level programming language, as suggested by Winograd [1979]. Though our intention is to teach a standard language, e.g. Pascal, not all users need continue on to this additional level of complexity. Those students that do not learn Pascal are limited to using the provided plans. The more a student learns about translating plans into code, the more directly they can express their intentions.

2. Programming Languages for Novices

The approach presented here 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 code. The BridgeTalk visual language presented here is 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:

- Above the loop — A programmer must declare a counter variable and initialize it to zero using an assignment statement.
- Inside the loop — The counter must be incremented, again using an assignment statement with a peculiarly non-algebraic construction:

```
Count := Count + 1.
```

Although an elementary concept, this is a very weird construction when viewed from outside the domain of programming. It requires a particular model of how computer memory is implemented in terms of extracting a value from a memory cell, incrementing the value, and storing the result back in the same cell. (Burstein [1986] discusses the assignment construct in detail.)

- 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 any problem actually being solved in the code.

Languages like Pascal are rooted in a programming model that is not closely related to the purpose and goals of the programmer using the language. Research into how novices learn programming confirms that the semantics of typical programming languages are not closely related to the way a typical novice understands a program [Bonar 1987]. 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 actual program code (see, for example, Eisenstadt et al. [1981], Mayer [1979], or Soloway and Ehrlich [1984].) Our approach, embedded in the design of the BridgeTalk visual language, allows students to explicitly represent their goals and describe how those goals interrelate.

2.1 Programming Plans: How Novices Learn to Program

Most programming texts teach students almost nothing about standard programming practices between the statement level and the fair' vague "structured design" level. That there are standard concepts and techniques for implementing common tasks is rarely mentioned. Typical tasks like "keeping a running total," "iterating down a list," and "searching a binary tree" are usually only covered implicitly through examples. (See Bonar and Weil [1986] for a collection of such concepts and techniques for introductory Pascal programmers.) We call these "standard concepts and techniques" *programming plans*, after the usage introduced by the Programmer's Apprentice Project [Rich and Shrobe, 1976; Waters, 1981].

In the last few paragraphs, we discussed the difficulties novice programmers have mapping task goals into code. Programming plans provide a representation of the goals of their task, a set of tools for translating those goals into actual code. BridgeTalk is intended to provide students with a way to program by manipulating plans.

Programming textbooks typically introduce a programming language by discussing the syntax and semantics of each statement type. Unfortunately, this approach exacerbates a common novice tendency to adopt a syntactic matching strategy

for problem solving. For example, physics students will often attempt to solve elementary mechanics problems by matching knowns and unknowns against standard formulae [Chi et al., 1981]. Their problem solving degenerates into a syntax-directed search with no understanding of the quantities being manipulated. Experts, in contrast, analyze a problem in terms of standard intermediate concepts and techniques from past experience. In physics, for example, these concepts and techniques include "component vectors", free body diagrams, and conservation of energy.

Programming novices exhibit syntactic strategies similar to those of the physics novices. In our video protocols of novice programmers [Bonar, 1985] we see novices working linearly through a program, choosing each statement based on syntactic features of the problem text or program code. Programming plans are exactly the concepts students need to step above the syntactic approach. Students can work on programming problems using standard approaches as encoded in plans.

A language that allows students to use programming plans has other advantages. Because the plans are high level, they allow the student to directly address interrelationships, in particular plan *merging*. Plan merging is necessary because programming plans typically have several facets that end up as dispersed lines of code. We saw this earlier with the counter plan example. When the counter increment must be placed inside of a loop body after the loop test, we are merging a facet of the counter plan with a facet of the loop plan. Errors in this plan merging process form a critical area of novice difficulty [Spohrer et al. [1985]. With an explicit plan representation, students can work directly with plans and plan interactions, without confusion about exactly how the merged plans will be turned into code.

2.2 Why Current Programming Languages are Unsuitable for Novices

The goals and constructs of standard programming languages are inconsistent with the needs of novice programmers. This can best be seen by examining the aesthetics used to judge standard programming languages: economy of expression, distrust of defaults and implicit behavior, emphasis on abstraction and abstraction tools, and efficiency on standard computer architectures. Here, we take up each point in detail.

2.2.1 Economy of Expression

Given similar functionality, programming languages with fewer constructs are almost always considered better then those with more. Pascal is considered an excellent programming language design because it managed to add substantial new functionality — user defined types — and embody a new approach — structured programming — in a language with substantially fewer constructs than its predecessor ALGOL 60. The history of LISP has been driven by language researchers looking for one abstract construct that can be used to efficiently implement a host of more

specialized constructs. Finally, one of the most common criticisms of the language Ada is its large size and firm opposition to "subsets".

Economy of expression makes sense when considering the implementation of a programming language, but seems to interfere with an important part of learning programming. As discussed above, a wealth of evidence now exists that expert programmers use a large collection of programming plans — standard templates and structures for accomplishing typical programming tasks. Experienced programmers effortlessly implement hundreds or even thousands of such plans [Waters, 1985] in programming languages with relatively few — on the order of 50 — constructs.

Novice programmers, on the other hand, have major difficulties implementing plans with programming language constructs. Evidence from our own work [Bonar and Soloway, 1985; Bonar et al., 1986] suggests that novice programmers bring a vocabulary of programming-like plans from experience with procedural specification in natural language. The translation of these natural language plans into the the narrow, low level constructs provided by a programming language is a critical and often insurmountable problem.

Acquiring the new plans that are essential to mastering programming is certainly difficult. This difficulty is compounded because novices not only need to acquire the plans, but they also need to learn how to translate the plans into a programming language. Because the natural language plans are rooted in real world day-to-day procedural specifications, it is very tough for a novice to translate them into a sparce and formal set of programming language constructs.

To illustrate the general problem, consider the following excerpt from an interview with an introductory Pascal student. She was writing a program that reads, sums, and counts inside a loop. She had written the following lines (line numbers are given for reference):

```
(1)     repeat
(2)        Read (New);
(3)        Sum := Sum + New;
(4)        Count := Count + 1
(5)     until Count > 100
```

The first author asked this student if line (3) was a "different kind of statements" than line (4). To our surprise, she stated that even though these statements "look alike," they were quite dissimilar. She went on to explain that line (3) "has something to do with something you are gonna ... take out of the loop with you." On the other hand, she explained, line (4) "keeps the loop under control." We contend the student has a deep understanding of the programming issues involved, even though she may have a weak

understanding of the specific Pascal constructs. While she has a firm grasp on the plans to be accomplished, she has a much weaker grasp on the language design constraint that would cause statements with such different purposes to have such a similar look.

We conclude that the emphasis on economy of expression in most modern programming languages is misplaced in designing languages for novice programmers. By looking for ever more abstracted ways to express similar procedural behavior, modern languages have excised most clues of goal and purpose that are essential to novice understanding of a program. The student discussed above would have much preferred a language with two constructs, "counter update" and "running total update", to Pascal which subsumes these (and many other) plans in the assignment statement. For this reason, BridgeTalk is a richer language than standard languages like Pascal. The richness in BridgeTalk expresses a catalog of standard plans.

2.2.2 Distrust of Defaults and Implicit Behavior

Another trend in the development of programming languages has been the systematic removal of defaults and implicit behavior. Whereas early languages such as FORTRAN and BASIC assumed properties and initial values for variables — for instance, based on the first letter of the variable's name — such assumptions are rare in modern languages. The trend to deemphasize default and implicit behavior is consistent with the trend toward economy of expression. Typically, defaults and implicit behavior add more complexity to a language. For example, a programming language designer would argue against a default value mechanism when standard assignment statements at the top of a program can accomplish the same end. Even where defaults reduce the number of tokens needed in a program — implicit typing of variables, for example — the defaults complicate a formal description of how the program runs.

As with economy of expression, the lack of defaults and implicit behavior makes standard programming languages much more difficult for novice programmers. While it is no extra work for an expert programmer to know to explicitly specify that a counter is to be incremented by one, it is a confusing burden to a novice. As expert programmers, we all can remember (or conjure up) a few circumstances where a problem is best solved by incrementing a counter by other than one. Even though such circumstances are exceedingly rare, we force every novice to explicitly specify this, as if a counter were commonly implemented in some other way.

Consider another example. Plans like "iterate down a list" are typically implemented with constructs that *begin* by testing for a condition that indicates we have *finished* the entire list. Readers with programming experience will need to consider for a moment to realize that this is very peculiar. First of all, it makes little sense to put an ending test at the beginning of the construct, before we've talked about

what the construct is to do. Second, it is strange that the list structure itself does not "know" where its end is, recognize the end, and take some appropriate default action to stop the iteration. Such examples of missing defaults are pervasive. We give novice programmers considerable extra work by forcing them to translate common plans into much lower level programming language constructs.

2.2.3 Emphasize Abstraction and Abstraction Tools.

Modern programming languages emphasize tools for abstraction. Current programming language design and development focuses on developing language constructs that capture and summarize structure and function for later use. For example, older languages provide control structures while more modern languages provide tools for interprocess communication, with the original control structures available as special cases. As another example, older languages focus on memory management while newer languages focus on data structuring and reuse of components.

As we have discussed above, non-programmers do not need tools for more abstract expression of programming plans; they need more direct ways to express the actual plans. The formal power of highly abstract constructs is in contrast to the pragmatic familiarity of programming plans. From the point of view of a procedural mechanism, an assignment statement is an abstraction that simplifies and subsumes a host of specific "changes to a variable," including, in particular, incrementing a counter. However, by stripping out the real world pragmatics that underlie programming plans, such abstractions make much of the programming process implicit and unavailable to novice programmers.

2.2.4 Efficiency on Standard Computer Architectures

A crucial constraint in most modern programming languages is that the language run efficiently on standard computer architectures. Pascal, for example, in its implementation of sets and rigid specification of ordering among program components, was intentionally tailored for efficiency of implementation. The designer of Pascal set the development of "implementations . . . which are both reliable and efficient on presently available computers" [Wirth, 1971] as his second of two principal aims for the language. While such an aim is obvious in a language for professional programmers, it is questionable in a language for novice programmers.

Many programming constructs are quite confusing to a novice programmer. For example, the assignment statement has peculiar semantics when seen from a perspective other than one that understands the basic working of registers and synchronous busses. In our daily experience it is rare that copying is simpler to specify than movement, as it is with assignment. Furthermore, in our daily experience we

rarely encounter values that are easier to destroy than to displace, again as it is with assignment.

2.3 How Can Visual Languages Help Novices

There are two basic reasons to turn to a visual language in a programming language for novices. First, a visual language provides the flexibility and expressiveness needed for a novice language to express a large vocabulary of programming plans. In our experience, a linear textual version of a plan-based programming language is quite unwieldy. Second, from a point of view of novice cognitive capability, a visual language is less likely to tax a novice's working memory. We discuss each of these points in detail.

2.3.1 Visual Languages for Expressing Plans

In BridgeTalk the emphasis is on identifying the formal plan components and expressing their interrelationships. As discussed above, 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 [Spohrer and Soloway, 1985]. BridgeTalk provides an environment where students can focus on plan interrelationships.

The various BridgeTalk constructs are implemented as icons that can be picked up, manipulated, placed, and connected together. The critical constraint that dictated this approach was the need for a language that allowed atomic plans with multiple connections of different types. The essentially linear nature of textual languages makes it 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. In fact, the current iconic representation was suggested by the confusion of students using earlier textual versions. (See section 4 where we discuss the formative evaluation of BridgeTalk.) Although the plans of BridgeTalk are not completely general, they do allow a student to focus on the pragmatic knowledge that stands above particular programming language implementations of that knowledge.

2.3.2 Visual Languages and Novice Cognition

Anderson [1985] has documented novice programmer errors which arise from overloading the student's working memory. The difference between novices and experts, they found, was the ability of experts to organize and structure their knowledge within working memory. Other work (cited earlier) points to plans as the kinds of structures used by experts. By providing plan icons and a way to interconnect those plans, we give students a structure to use in organizing their understanding and problem solving.

2.4 Design Goals For BridgeTalk

There are four objectives for BridgeTalk:

(1) Allow users to "connect" to their informal (primitive) plans.

(2) Support novice programmers in learning a "vocabulary" of programming plans.

(3) Support novice programmers in learning how to implement plans with a standard programming language.

(4) Support the use of plan-like composition of programs.

We take up each of these goals separately.

2.4.1 Connecting to Informal Plans

Unless novices can recognize how their own understanding of plans fits into the programming environment they face, they will find it impossible to formulate correct solutions. Novice programmers have a large vocabulary of informal plans based on experience with step-by-step procedures in everyday life. We want BridgeTalk to connect to and build on this understanding. A subset of the plan icons in BridgeTalk must connect to the collection of naive plans brought in by programming novices.

This goal yields several benefits. First, there is the obvious benefit that students have at least some familiarity with the system, even before their first use. Furthermore, the informal plans brought in by a novice are typically a simplified form of plans that are central for experienced programmers. Finally, a simplified novice plan forms a good starting place for understanding the related set of richer, more formally specified expert plans.

Note that we are not arguing that a novices' collection of naive plans is sufficient for sophisticated programming. Instead, we are recognizing the importance of giving a beginner an anchor point in a new domain. We are very clear that experienced programmers have a much richer set of programming plans than a novice. We are also clear that the plans of an expert programmer are more fully articulated and elaborated than the plans of a novice. Nonetheless, we see great value in supporting the growth of a novice from primitive plans that are already known.

2.4.2 Developing an Explicit Vocabulary of Programming Plans

Although we begin with a student's informal plans, our ultimate goal is that a student leaves a programming course with a rich set of highly articulated programming plans. Even in a language with no explicit plan structure, e.g. Pascal, successful students do leave a course with a rich vocabulary of plans. With BridgeTalk, we provide a mechanism for this vocabulary to be an explicit part of programming instruction.

An explicit vocabulary of plans provides an opportunity to formally specify programming curriculum. Although elementary programming courses often discuss issues of design, abstraction, and structure, this is rarely done with any rigor. We suggest that the rigor is missing because there is no adequate vocabulary with which to discuss programming design. Typical programming constructs operate at too low a level. At the other end of the spectrum, formal approaches to correctness offer little to a student still attempting to understand the design space of programming.

Essential to the development of a plan vocabulary are techniques for extending and modifying plans in a systematic way. In BridgeTalk we provide techniques for extending and specializing a particular plan. Students just beginning with BridgeTalk can use the plans provided with the system. More experienced students can develop their own plan vocabulary.

2.4.3 Implementing Plans With a Standard Programming Language

If a student desires, BridgeTalk should support that student in learning to program in a standard programming language such as Pascal. In general, this means that the plan formalism must provide a way to derive a conventional program from any plan program. BridgeTalk provides a mechanism whereby students can successively "look inside" a plan, and at each level reveal an implementation closer and closer to typical programming constructs. This approach takes advantage of the strengths of both plans and programming constructs. On one hand, the plans abstract away from particular programming constructs, specifying higher level goals. On the other hand, the programming constructs are more general purpose than the plans.

The system is layered for interrelated implementation and pedagogical reasons. By providing various layers, we can take advantage of shared structure between various plans. For example, the counter plan is a specialization of the running total plan, and is implemented that way. Pedagogically, it is valuable for students to see the shared structure inherent in the various plans. By using multiple layers, we need only show the plan components required to illustrate the commonality, without showing all the underlying detail. So, for example, the counter plan can be described as a specialization of the running total plan where the update is always one. Note that most details of the plan implementations need not be understood to understand their commonality.

2.4.4 Supporting Plan-like Composition of Programs

The final design goal is that BridgeTalk support programming by plan composition. To support this goal each particular plan has a single icon. We feel this is crucial if we are to allow a student to focus directly on plan interconnection separate from other issues. As discussed above, one of the crucial difficulties of a programming language like Pascal is that simple plans can end up expressed in several dispersed

lines of code. Our goal for BridgeTalk is a representation that maintains a plan as an atomic element.

Even though the plans are atomic, we must be able to distinguish between components contained within the plan. For example, a looping plan needs to have separate subcomponents that represent a test, a body, and a modification that can change the value of the test.

3. A Visual Language for Novices

3.1 Informal Overview

A key design constraint of the BridgeTalk language is that each particular plan has a single icon. We feel this is crucial if we are to allow a student to focus directly on plan interconnection separate from other issues. As discussed above, one of the crucial difficulties of a programming language such as Pascal is that simple plans can end up expressed in several dispersed lines of code. Our goal for BridgeTalk is a representation that maintains a plan as an atomic element.

Consider the following example programming problem, which is used in our discussion. We call this the Ending Value Averaging Problem. The 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.*

The BridgeTalk plan icons used for the Ending Value Averaging Loop plan are shown in Figures 1 and 2. The general metaphor of the plan language is that of puzzle pieces being fit together. Plans with similar shapes have similar kinds of roles in a program. Plans that express a sequence of values in a variable, called "variable plans," are shown as squares (see Figure 1). Each of these plan icons embodies the entire semantics of a particular plan.

The Counter plan, for example, keeps and shows a value (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 a value (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 the 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 1 shows a Compute plan. The Compute plan uses values supplied by variable plans (the connection to a variable plan is

described below), and an operation selected by the user. 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.

Figure 2 shows a New Value Controlled Loop plan, one of the most complex plan icons in BridgeTalk. There are four components of the New Value Controlled Loop plan:

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 a traversal of a data structure.

end of loop condition — A test to determine when to exit the loop. Note that this test is constrained to do a particular kind of test: checking 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.

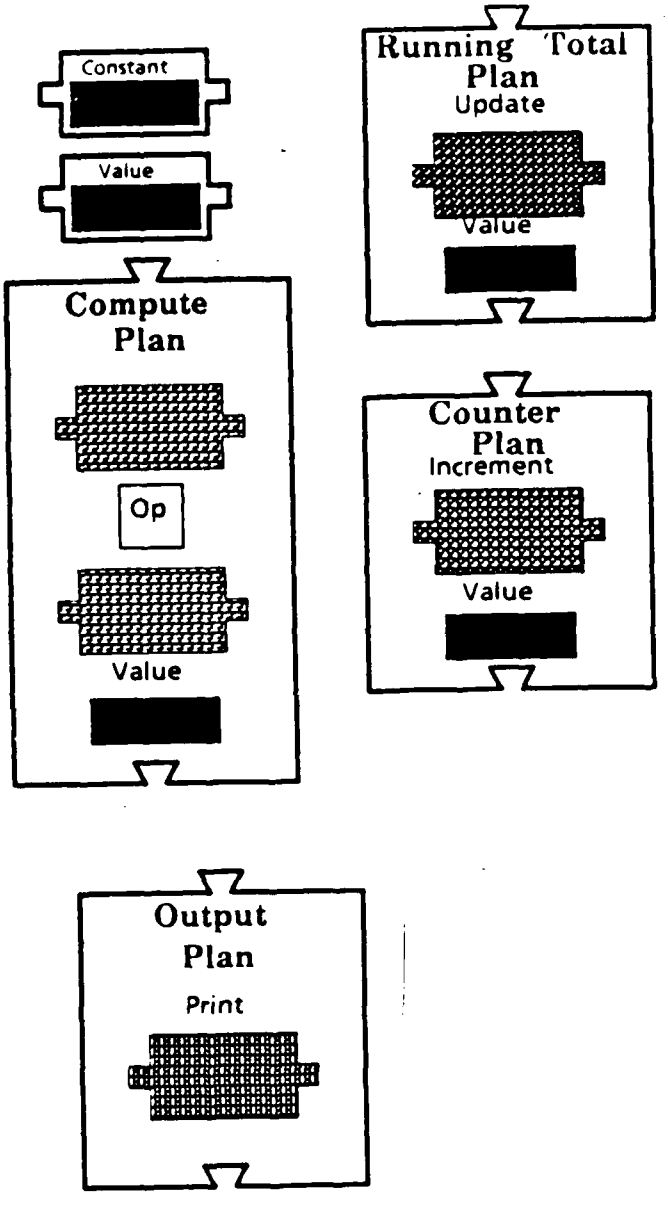
body of loop — A series of plan icons to be executed in the body of the loop.

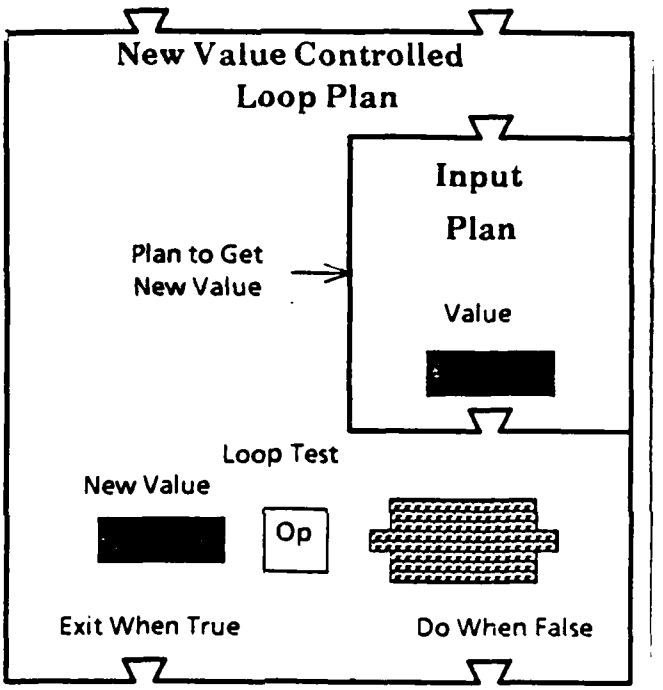
actions to be performed after loop is complete — A series of plan icons to be executed when the loop completes.

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 in a standard language. (Soloway, et al. [1983] presents detailed data on this complexity.)

Figure 3 shows a BridgeTalk solution to the Ending Value Averaging problem. The focus of BridgeTalk 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 BridgeTalk 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 3 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 (a Value plan and corresponding hole are shown in Figure 4.) Value plans are created by selecting the box labeled "Value" within a plan that can produce a value. Figure 5a shows a Running Total plan whose value box has just been selected. Such a selection creates a Value plan icon that is


Constant
Value
Compute Plan
Op
Value
Running Total Plan
Update
Value
Counter Plan
Increment
Value
Output Plan
Print


New Value Controlled
Loop Plan
Input
Plan
Plan to Get
New Value
Value
Loop Test
New Value
Op
Exit When True
Do When False

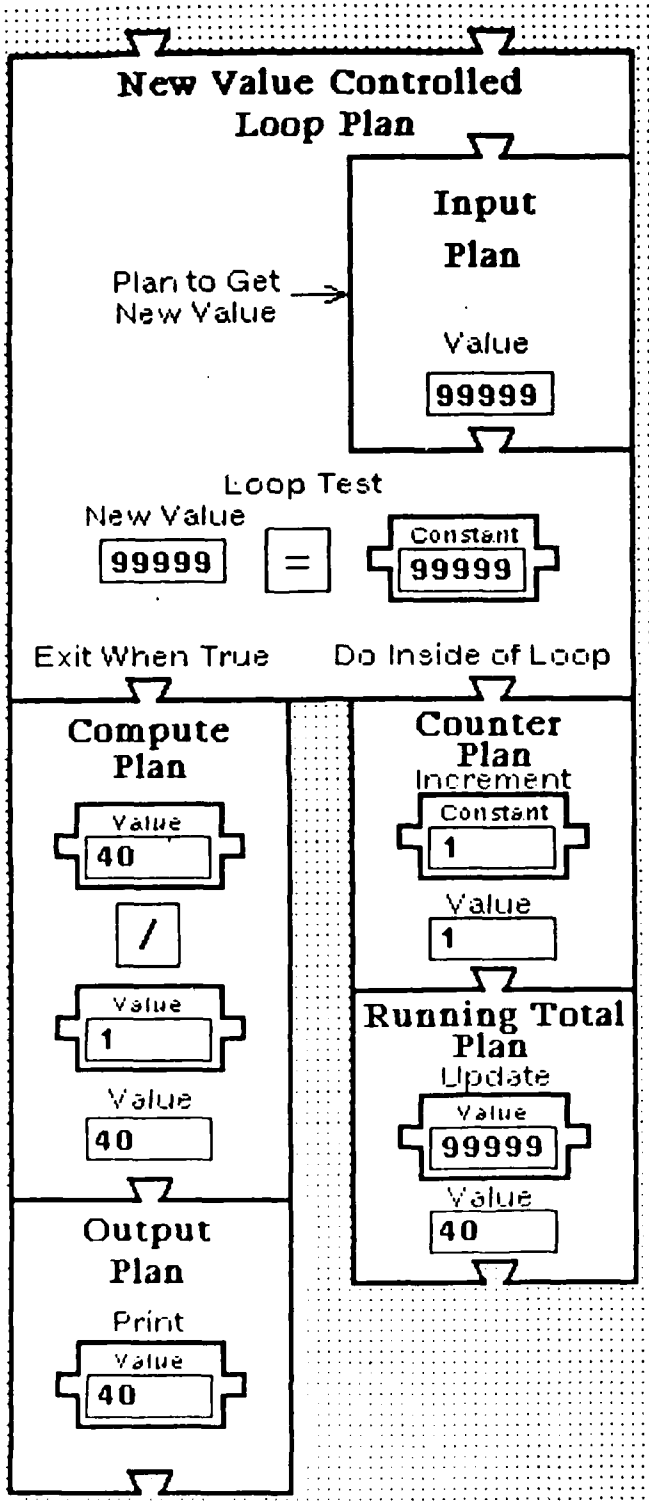

New Value Controlled Loop Plan
Input Plan
Plan to Get New Value
Value
99999
Loop Test
New Value
99999
=
Constant
99999
Exit When True
Do Inside of Loop
Compute Plan
Value
40
/
Value
1
Value
40
Counter Plan
Increment
Constant
1
Value
1
Running Total Plan
Update
Value
99999
Value
40
Output Plan
Print
Value
40

FIGURE 3

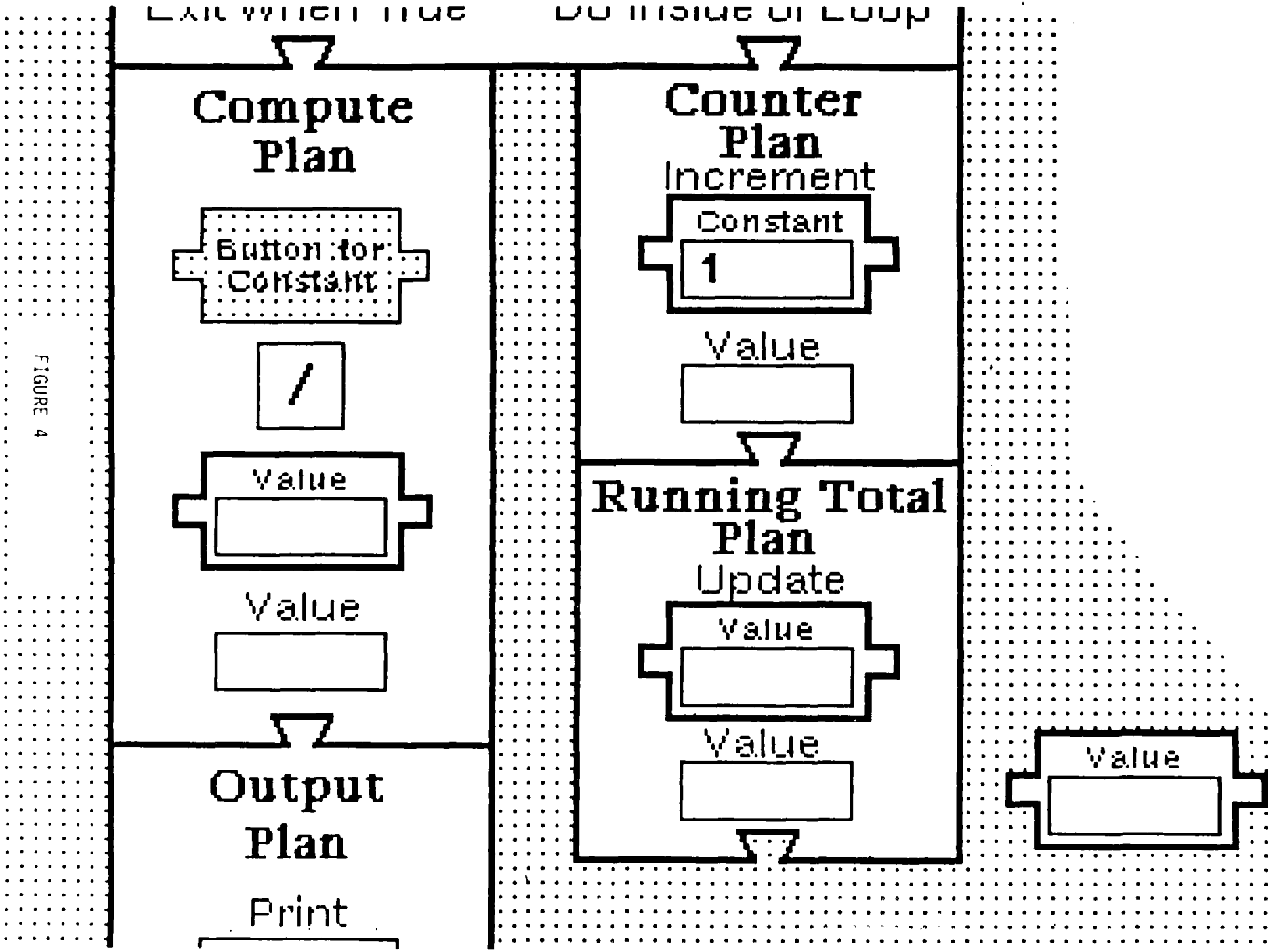

Exit when true
Do inside of Loop
Compute Plan
Button for Constant
/
Value
Value
Output Plan
Print
Counter Plan
Increment
Constant
1
Value
Running Total Plan
Update
Value
Value
Value

FIGURE 4

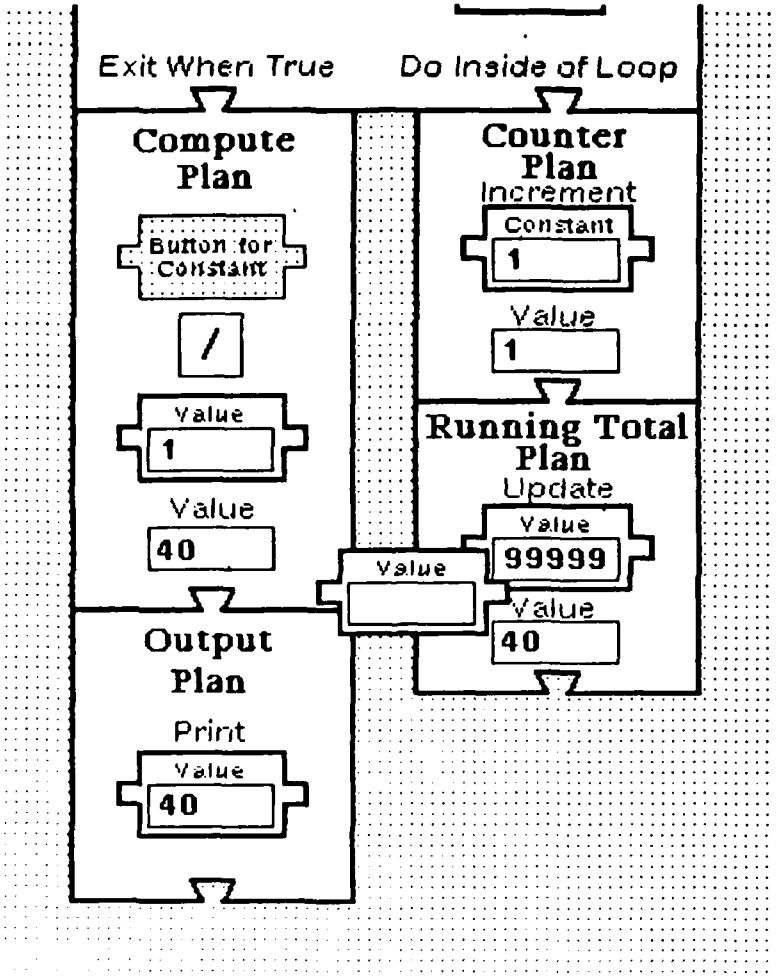

Exit When True
Do Inside of Loop
Compute Plan
Button for Constant
/
Value
1
Value
40
Output Plan
Print
Value
40
Counter Plan
Increment
Constant
1
Value
1
Running Total Plan
Update
Value
99999
Value
Value
40

FIGURE 5A

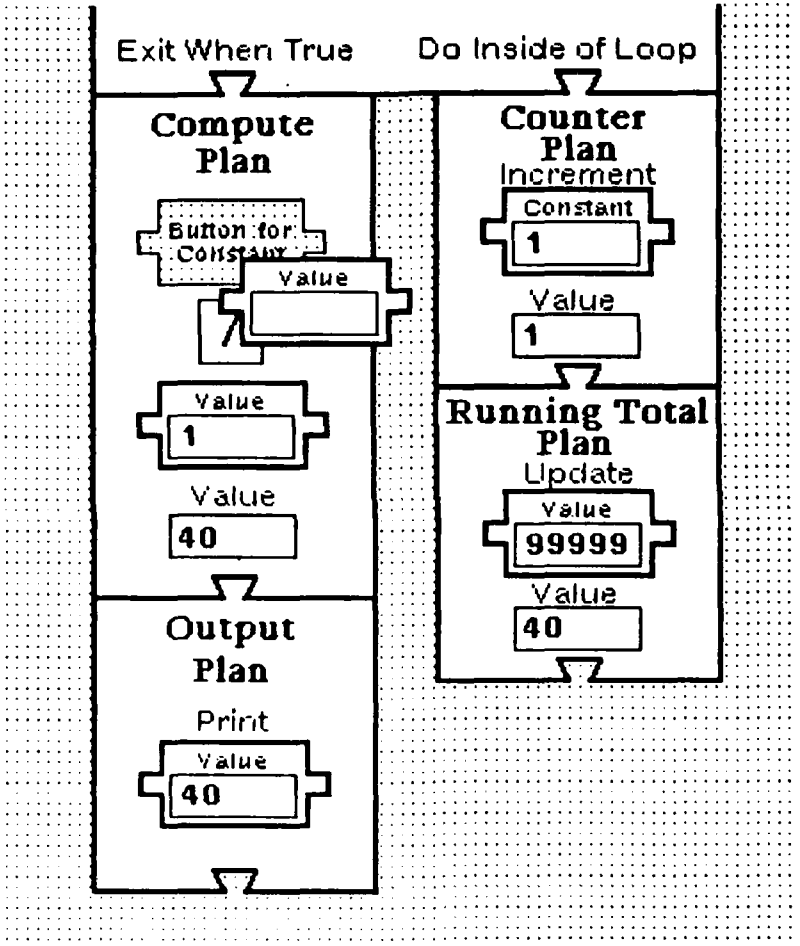

Exit When True
Do Inside of Loop
Compute Plan
Button for Constant
Value
Value
1
Value
40
Output Plan
Print
Value
40
Counter Plan
Increment
Constant
1
Value
1
Running Total Plan
Update
Value
99999
Value
40

FIGURE 5B

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. Figure 5b shows the Value plan from the Running Total plan being placed in a hole of the Compute plan.

We have made the decision that BridgeTalk expresses control flow in a more direct way than data flow. Control flow connections have a direct visual expression. If plan A is executed before plan B, the tab on the top of B's icon fits into the slot on the bottom of A's icon. Data flow connections, while specified directly by dragging an object from one plan to another, leave a much more subtle visual clue. In theory, we could have designed the plan language to emphasize visual expression of data flow instead of control flow, or even to emphasize either, depending on a mode selected by the user. One earlier plan language had a visual representation for both control and data flow simultaneously, but proved to be too complex and unwieldy for our students. We chose to favor control flow over data flow in order to best match Pascal, the current "default" novice programming language.

3.2 Formal Description

This section describes a formal representation for plans that forms the basis for our visual programming environment to aid novices in learning to program. Our approach allows several representations of a problem solution. The user is able to navigate from one representation to another in order to obtain different points of view on the solution. The problem solution can be modified from any one of these representations, or levels. The solution is *descriptive* at the highest level, *prescriptive* at the lowest. This top level uses icons to identify specific plans that the user links together appropriately. At the lowest level is source code for a textual, procedural language, such as Pascal.

3.2.1 Plan Representation

The plan formalism is based on object-oriented programming. For each type of plan there is a class that specifies the local data and operations of that plan. Instances of a plan class can then be created, each with their own copy of the local data. The user organizes these instances into a particular execution order using abuttment, embedding, and merging.

Each plan is represented with up to four parts:

ParentClass — this section provides a link indicating a hierarchical relationship among plans for purposes of taxonomy and inheritance. Each plan is a member of exactly one class. If this section is missing, there is no parent class to the plan.

Slots — each plan can have zero or more slots, which provide one of two types of entities - data or plan links. The data slots are untyped at this stage of the system, and contain a single value. One slot can be distinguished as the "value" slot for this plan, meaning that the plan acts like a function and creates a single value that can be used elsewhere. The plan links provide a method for referring to other plans for purposes of abuttment and embedding.

Initialization — this optional section contains "executable code" that is performed once when the plan is first accessed. Whenever a plan is entered, if its initialization section has not been previously executed, it is fired. As a special condition, when a loop plan is entered, it fires its initialization section and the initialization sections of all plans contained in its body.

Execution — this optional section contains "executable code" that is performed whenever the plan is accessed.

The code given in the Initialization and Execution sections is expressed in a simple pseudo-code which creates the required control structures. This code can be easily translated into a standard programming language such as Pascal, given the plan representations and the pseudo-code. There are sufficient constraints on the representation to make this a straightforward process. It can be done incrementally, allowing for user modification of the code. The user will have access to this code, allowing him/her to examine the way a plan is implemented in a standard programming language.

Figure 6 gives an example of the above scheme. This example forms a hierarchy of refinement for the notion of a counter. At the top-most level is a Loop Action plan. The lack of a ParentClass designation simply means that this plan inherits from no other class. In addition, there are no Slots, Initialization, or Execution sections associated with this type of plan. Its purpose in the taxonomy of plans is to indicate that any plan decended from the Loop Action class must be used within the body of a loop.

Next, the Running Total plan contains three slots, all data slots. The angled brackets (< >) indicate data that has no pre-specified value. The slot Total has been prefixed with "value." as an indication that it is the distinguished slot that will be taken as the value of this plan. Finally, the Initial slot has been given an explicit value of zero. The asterisk indicates that this is a default value, and can therefore be changed by the programmer.

LOOP ACTION PLAN

RUNNING TOTAL PLAN

PARENTCLASS:
LOOP ACTION

SLOTS:
value.TOTAL -- < >
ADDEND -- < >
INITIAL -- 0*

INITIALIZATION:
TOTAL <--- INITIAL

EXECUTION:
TOTAL <--- TOTAL + ADDEND

CONSTANT RUNNING TOTAL PLAN

PARENTCLASS:
RUNNING TOTAL

SLOTS:
ADDEND.INCREMENT -- (CONSTANT)

COUNTER PLAN

CLASS:
CONSTANT RUNNING TOTAL

SLOTS:
TOTAL.COUNT -- < >
ADDEND.INCREMENT -- 1*

FIGURE 6

ENDING VALUE AVERAGING LOOP PLAN

```
SLOTS:
        TOTAL -- (RUNNING TOTAL)
        COUNT -- (COUNTER)
        LOOP -- (SENTINEL LOOP: BODY <-- [TOTAL,COUNT])
        value.AVERAGE -- (AVERAGE: DIVIDEND <-- TOTAL; DIVISOR <-- COUNT)
        OUTPUT -- (OUTPUT: OUT <-- AVERAGE; MESSAGE <-- "The average is ")

EXECUTION
        LOOP <= = Execute
        AVERAGE <= = Execute
        OUTPUT <= = Execute
```

SENTINEL LOOP PLAN

```
PARENTCLASS:
        NEW VALUE CONTROLLED LOOP

SLOTS:
        NEWVALUE -- (INPUT: MESSAGE <-- "Enter a new value")
        TESTVALUE.SENTINEL -- (CONSTANT)
        WHENTOHALT -- ' = '*
```

NEW VALUE CONTROLLED LOOP PLAN

```
SLOTS:
        NEWVALUE -- < >
        TESTVALUE -- < >
        WHENTOHALT -- < >
        TEST -- (TEST: TESTVALUE1 <-- NEWVALUE; TESTVALUE2 <-- TESTVALUE;
             RELATIONALOPERATOR <-- WHENTOHALT)
        BODY -- < >

EXECUTION:
        Loop
                NEWVALUE <= = Execute
                TEST <= = Execute
                If value.TEST is False then Exit
                BODY <= = Execute
        Endloop
```

FIGURE 7, part 1 of 3

COMPUTE PLAN

SLOTS:
value.RESULT -- < >
OPERAND1 -- < >
OPERAND2 -- < >
OPERATOR -- < >

EXECUTION:
RESULT <-- OPERATOR(OPERAND1,OPERAND2)

AVERAGE PLAN

PARENTCLASS:
COMPUTE

SLOTS:
RESULT.AVERAGE -- < >
OPERAND1.DIVIDEND -- < >
OPERAND2.DIVISOR -- < >
OPERATOR -- '/'

TEST PLAN

PARENTCLASS:
COMPUTE

SLOTS:
OPERAND1.TESTVALUE1 -- < >
OPERAND2.TESTVALUE2 -- < >
OPERATOR.RELATIONALOPERATOR -- < >

INPUT PLAN

SLOTS:
value.IN -- < >
MESSAGE -- < >

EXECUTION

FIGURE 7, part 2 of 3

OUTPUT(MESSAGE) < = = Execute
INPUT(IN) < = = Execute

OUTPUT PLAN

SLOTS:

value.OUT -- < >
MESSAGE -- < >

EXECUTION:

OUTPUT(MESSAGE) < = = Execute
OUTPUT(OUT) < = = Execute

FIGURE 7, part 3 of 3

The Initialization and Execution sections give typical "code" for how a running total works. As a member of the Loop Action class, the Running Total plan will naturally be embedded in a loop of some kind, so there is no need to explicitly specify the loop itself here.

The Constant Running Total plan is a specialization of the Running Total plan. The only difference is that the addend is a constant of some sort, so that this plan sums a single value over and over. The one slot in this plan shows two additional notations. First, the slot Addend that was used in the Running Total plan has been renamed to Increment. This renaming is indicated syntactically by using a concatenation of the two names. The renaming makes it easier to construct meaningful messages for the user. Second, this slot has been redefined to include a link to another plan, namely the Constant plan which asks the user for a value for the constant to be summed.

Finally, the Counter plan shows a further refinement of the Addend.Increment slot to be a specific value of 1 (default). In addition, the slot Total that was inherited from some parent class (in this case, from Running Total) has been renamed to Count. The renaming does not affect the designation of this slot as the "value" slot, however.

3.2.3 Inheritance

Each plan must specify a parent class explicitly, unless there is no parent. This provides a linkage for the inheritance mechanism.

Slots are inherited from all predecessors, but may be renamed or redefined. Note in the example given in Figure 6 that the Counter plan has inherited (and renamed) the Total slot from the Running Total plan, although the intervening plan (Constant Running Total) does not explicitly reference it. In addition, the Counter plan inherits Total's designation as the distinguished value of the Running Total plan.

The Initialization and Execution sections can also be inherited. In this case, both are inherited by successors from the Running Total plan. The renaming and redefining mechanisms allow us to think of the execution of the Counter plan as

Count <-- Count + 1.

In addition, it allows us in conversation with the user to describe the "1" as an "increment" rather than as an "addend."

3.2.4 A More Complex Example

Figure 7 shows the representation of the Ending Value Averaging Loop plan and its components. This plan is used to calculate the average of a running total of values that are entered by the user.

Some additional notation occurs in this example. Note that it is possible to pass values to linked plans. For instance, in the case of the Average slot, the Average plan is passed the values of Total and Count with which the Average plan will compute the average. These values are substituted for the slots Dividend and Divisor, respectively, in the Average plan. The Output slot sends this average to the Output plan.

The Execution section of the Ending Value Averaging Loop plan shows how slots can be executed by sending them an Execute message. The meaning of this code is that the slots identified are to be executed sequentially, forming an abuttment of the plans contained in the named slots.

Note that the Loop slot contains a reference to the Sentinel Loop plan. This Sentinel Loop plan is a subclass of the New Value Controlled Loop plan. The Sentinel Loop plan does not have any executable sections, but does define three new slots that will be used during execution. In addition, even though the Sentinel Loop plan itself does not contain a slot called Body, as referenced in the Loop slot of the Ending Value Averaging Loop plan, it inherits such a slot from its parent class, the New Value Controlled Loop plan.

The Body slot requires some additional explanation. Generally, a slot being used as a parameter contains the name of a single value or slot. In the case of Body, however, several slots (and, therefore, plans) can be abutted to form, for instance, the body of a loop. The Body slot, then, acts similar to a BEGIN-END block in Pascal. In this example, the body of the Ending Value Averaging Loop plan is identified as being the Total slot (and, consequently, the Running Total plan), followed by the Count slot (linking to the Counter plan.

Finally, note that the code for performing the actual looping action appears only in the New Value Controlled Loop plan. The Ending Value Averaging Loop plan and the Sentinel Loop plan leave this detail to another level so that it can been hidden from the user to some extent. In this case, this executable code is inherited by the Sentinel Loop plan, where the values given in its slots (which are references to other plans) are used to "fill in the blanks" of the code. In this way, a subclass plan can redefine slots that have been originally defined in its parent plan.

The Body of the loop in the executable code given in the New Value Controlled Loop plan actually comes from the Ending Value Averaging Loop plan. In this way, the actual contents of the loop body can be customized to fit the requirements of particular plans. In this case, it is the Ending Value Averaging Loop plan that should define that the body of the loop contains Running Total and Counter plans. The Sentinel Loop plan is not responsible for this detail, since its only concern should be how to construct a sentinel-valued loop.

3.2.5 Meeting the Goals

The notions of programming plans and informal natural language plans are fundamental to this system. The "connection" between these two types of plans obviously must be provided in a natural, easily understood manner in order to be obvious to a novice. Using plans already is a first step toward providing an intuitive environment. It will be the responsibility of the interface to provide the rest of the connection that is required. This is one of the main reasons why a visual programming environment is desireable. The interface would naturally need to include appropriate icons for the available plans, a graphics editor for creating a program, execution displays, etc. The rest of the discussion in this section assumes a suitable interactive interface, much like the one currently available in the Bridge programming tutor [Bonar et al. 1987].

Initially the novice would use plans at the highest possible level, constructing programs by selecting from a menu of plan icons and connecting the icons together using abuttment, embedding, and merging. For instance, he/she might use the Ending Value Averaging Loop plan with only the understanding that such a plan would somehow total up a series of numbers and compute their average. This can be seen simply by looking at the representation for Ending Value Averaging Loop plan given in Figure 7, for instance, without referring to any of its sub-plans. Exactly how this will be presented to the user visually is not restricted by the representation given above. The actual formal representation could be one of the alternate representations available to the user, or could be just the base representation for all others without being visible itself.

A curious student, however, would probably want to know more about the plans identified in the slots of the Ending Value Averaging Loop plan. In such a case, the user could ask the system to navigate the hierarchy so that he/she could view, for instance, the Sentinel Loop plan. In this way the user can study the taxonomy of the plans, and learn something about plan construction.

Further interest might lead the user to examine the code that implements the Sentinel Loop plan (as inherited from the New Value Controlled Loop plan). This can be derived from the representation given in Figure 7. Exactly what form this will take is yet to be determined. Possibilities include diplaying the pseudo-code shown in Figure 7, as well as an implementation in a standard language such as Pascal. In addition, it is possible to allow the user to modify this code directly, rather than relying entirely on the iconic representations of the plans. This lets the user -- as his her skill grows -- to define his/her own plans. Selective execution of code is also possible with the system. This all helps the user to understand programming plans and their peculiar vocabulary.

The navigation described above takes place in two distinctly different ways. In moving from the Ending Value Averaging Loop plan to viewing the Sentinel Loop plan, the user is scanning the details of the plan itself and its various component parts. Figure 8 shows a graphical representation of the Ending Value Averaging Loop plan and its components, as developed from the plan description of Figure 7 with all inheritences resolved. This figure has fairly standard connotations of hierarchy of the plans involved. It also contains some indication of plan construction -- the multi-branched lines indicate abuttment of plans, as in the case of the Loop, Average, and Output slots of the Ending Value Averaging Loop plan, and the Running Total and Counter plans (via the Total and Count slots, respectively) contained in the Body of the Sentinel Loop. This figure shows the plans involved with the Ending Value Averaging Loop mechanism at their highest level, which is essentially descriptive.

A second way that the plans might be navigated, changing to a different point of view, is through the Class hierarchy. This was somewhat implied by viewing the code implementing the Sentinel Loop plan above, inherited from the New Value Controlled Loop plan. Figure 9 shows a more specific example. Counter is shown to be a sub-class of the Constant Running Total plan, which in turn is a sub-class of the Running Total plan, which is a Loop Action plan. The slots of the Running Total plan are inherited by the lower levels. In this case, the Initial slot is inherited without modification by lower levels.

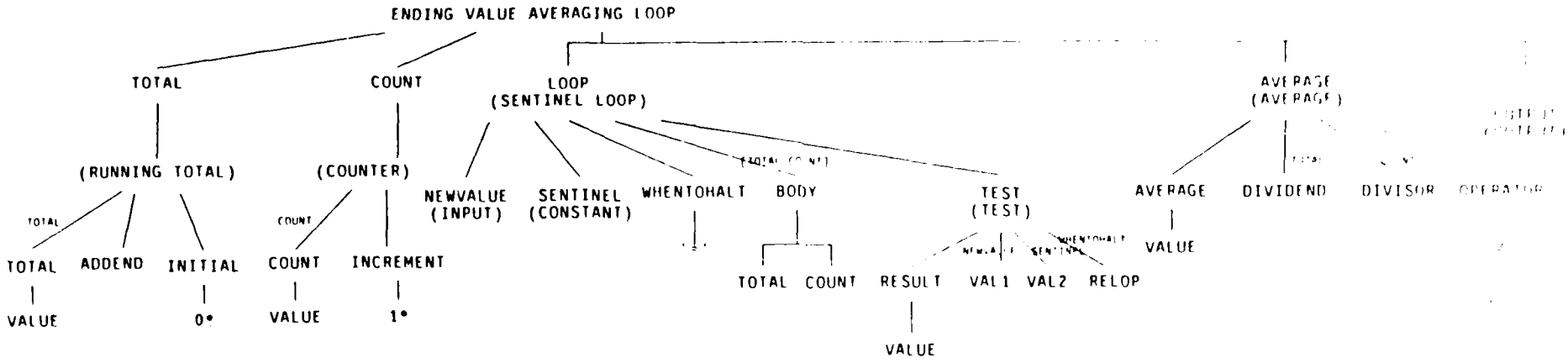
The Constant Running Total plan is shown redefining and renaming the Addend slot from its predecessor. The Counter plan also redefines this slot further. The Counter plan also renames the Total slot from above.

Navigating this hierarchy is considerably different than in the previous case shown in Figure 8. When moving from the Counter plan to the Constant Running Total plan, to the Running Total plan, we are moving from more specific levels to more general ones. It is more specific to say that we are counting, with an increment of 1, than to keep a running total of some unspecified addend.

This hierarchy can be considered as orthogonal to the one shown in Figure 8. It represents additional detail that the user can explore for a deeper understanding of programming. Traversing these levels also lets users explore the plan taxonomy on another level. In this way the user gains a more sophisticated point of view of the programming environment.

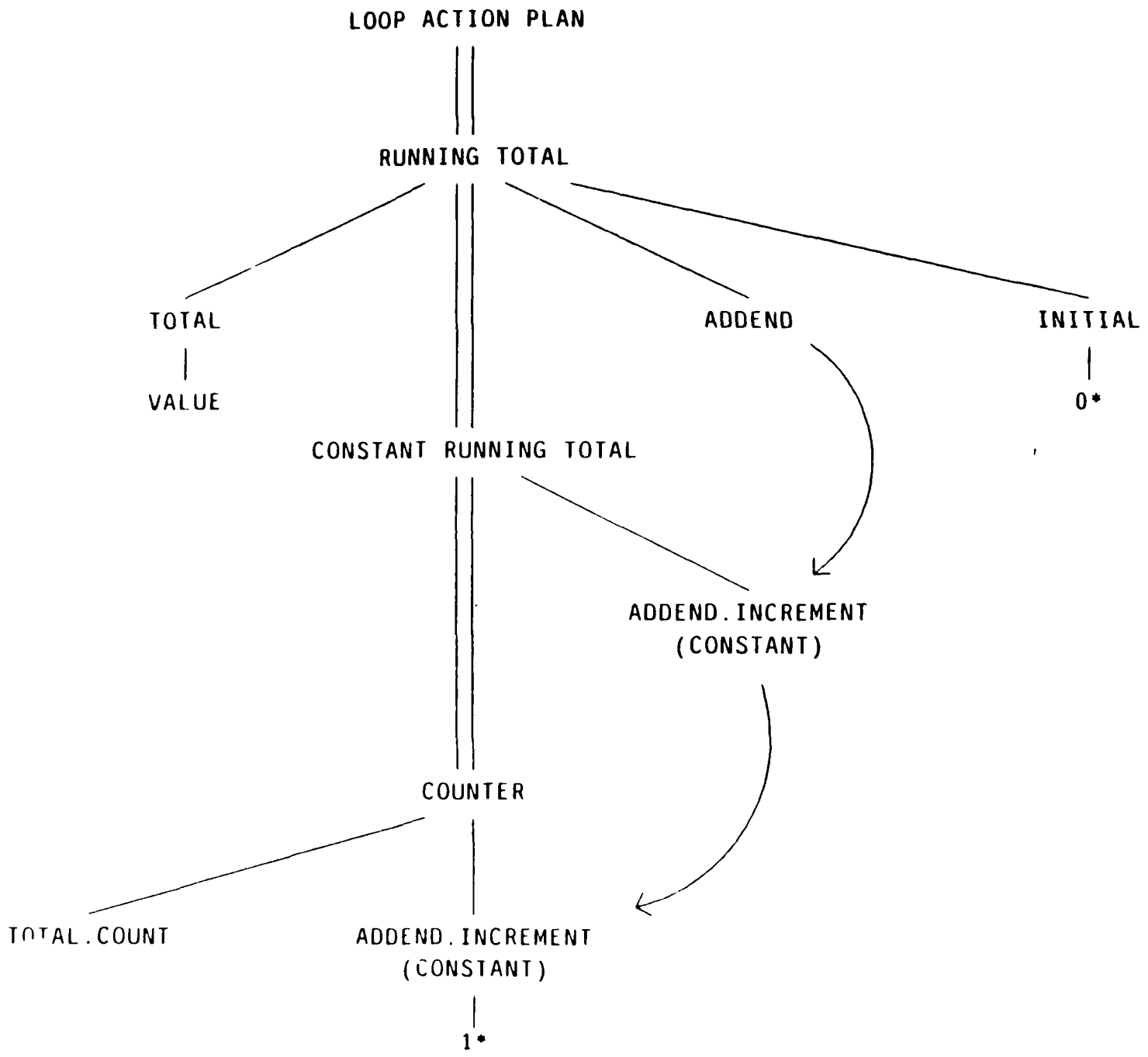
3.3 Implementation

A detailed discussion of the implementation of the plan language discussed above is beyond the scope of this article. The current prototype is implemented in LOOPS, an



*DEFAULT

FIGURE 8


LOOP ACTION PLAN
RUNNING TOTAL
TOTAL
VALUE
ADDEND
INITIAL
0*
CONSTANT RUNNING TOTAL
ADDEND.INCREMENT
(CONSTANT)
COUNTER
TOTAL.COUNT
ADDEND.INCREMENT
(CONSTANT)
1*
*DEFAULT

object-oriented extension to Interlisp-D on Xerox 11xx workstations. BridgeTalk is embedded within the Bridge intelligent tutoring system for teaching Pascal [Bonar, et al. 1987]. The visual language is developed in Chips, a tool for building direct manipulation interfaces developed at the University of Pittsburgh's Learning Research and Development Center [Corbett and Cunningham, 1987]. Chips was particularly valuable in allowing the rapid development of prototype visual interfaces for testing with students.

BridgeTalk is the result of an indepth cycle of research, design, testing with students, and evaluation. As is documented in detail in the next section, BridgeTalk has been through six distinct generations of that cycle. The first two generations each took six person-months of program development time, along with one person-month each of empirical testing. The last four generations were each developed in less than three person-weeks, using the Chips interface design tool. Each of these last four generations also took about a person-month of empirical testing.

4. Formative Evaluation and Design History

BridgeTalk has not been systematically tested against a standard programming language like Pascal in a classroom situation, though such a test is planned. BridgeTalk is the result, however, of a long process of formative evaluation and development. BridgeTalk has been extensively tested with students through a number of different versions. In this section we summarize the design history of BridgeTalk over the course of 6 generations.

4.1 Generation 1

Figure 10 shows an Ending Value Averaging solution in the original version of BridgeTalk. The student uses a mouse to "pick up" and place each element in its proper place in the developing program.

There were two main objectives in this initial version of BridgeTalk. First, the student was to gain a better understanding of the flow of control implied by the various plans, and the interrelationships of the various plans. In the case of the loop construct, an icon was supplied that was meant to show graphically what the looping entailed. Two other icons, for input and output, were also supplied in an attempt to illustrate their function.

The second objective was to begin to show a separation of the various roles each plan may have. For instance, in the language of Figure 10 the user is faced with the need to identify where the initialization of a variable should occur. In the case of the Counter Variable plan, note the initialization and increment roles in separate locations in the figure.

This version of BridgeTalk was deemed unsatisfactory for several reasons. First, the user was required to develop an entirely new vocabulary in terms of the specific plan phrases given. The phrases used were somewhat cryptic, making identification of the proper plan difficult.

A second problem was with the icons. Aside from their crudity, they were not particularly indicative of the function represented. The use of icons was also limited to only three constructs, input, output, and looping. In addition, those icons served no dynamic function, but were only somewhat abstract graphics used to highlight the particular plans. The main reason these specific three plans were chosen for graphical representations is that they seemed easily represented. It simply was not clear how to represent the other plans neatly using this sort of graphics.

A third problem was that this new representation of a problem solution was still static. This meant that the user still had to imagine how this solution actually would work. The static nature of the solution representations required too much computing skill from the user.

4.2 Generation 2

Version 2 of BridgeTalk was more data oriented. More complete descriptions of the various roles of each plan were given. Figure 11 shows a selections of plans from this second version. Plans are represented by boxes containing slots to be filled. Each slot represents a role, represented as a tile, of some other plan. This helped make each plan more self-explanatory. The underlying notion was that a plan could be more systematically treated as a frame with slots. The control structures became frames, and data were slots.

Figure 12 shows how a solution to the Ending Value Averaging problem was formed. Either an entire plan or just its slots could be used in the final solution. In this example, the Control Loop with Sentinel plan is used in its entirety. Its open slots are filled with tiles from other plans. In the case of the Count How Many plan, only its tiles are used, not the entire box identifying the plan.

While this approach does indeed make the data objects being manipulated more identifiable, and does give more explicit help to the user in terms of explaining the plans, there are still many problems. To begin with, the layout employed implies a fairly loose interpretation of plans by the user. In some cases, the entire plan's box is used. In others, only the tiles are used. In addition, sometimes these tiles are used to fill in slots of other plans, while at other times they are simply placed individually in the figure, with no direct connection to any other object in the figure. How does a novice interpret these differences of use?

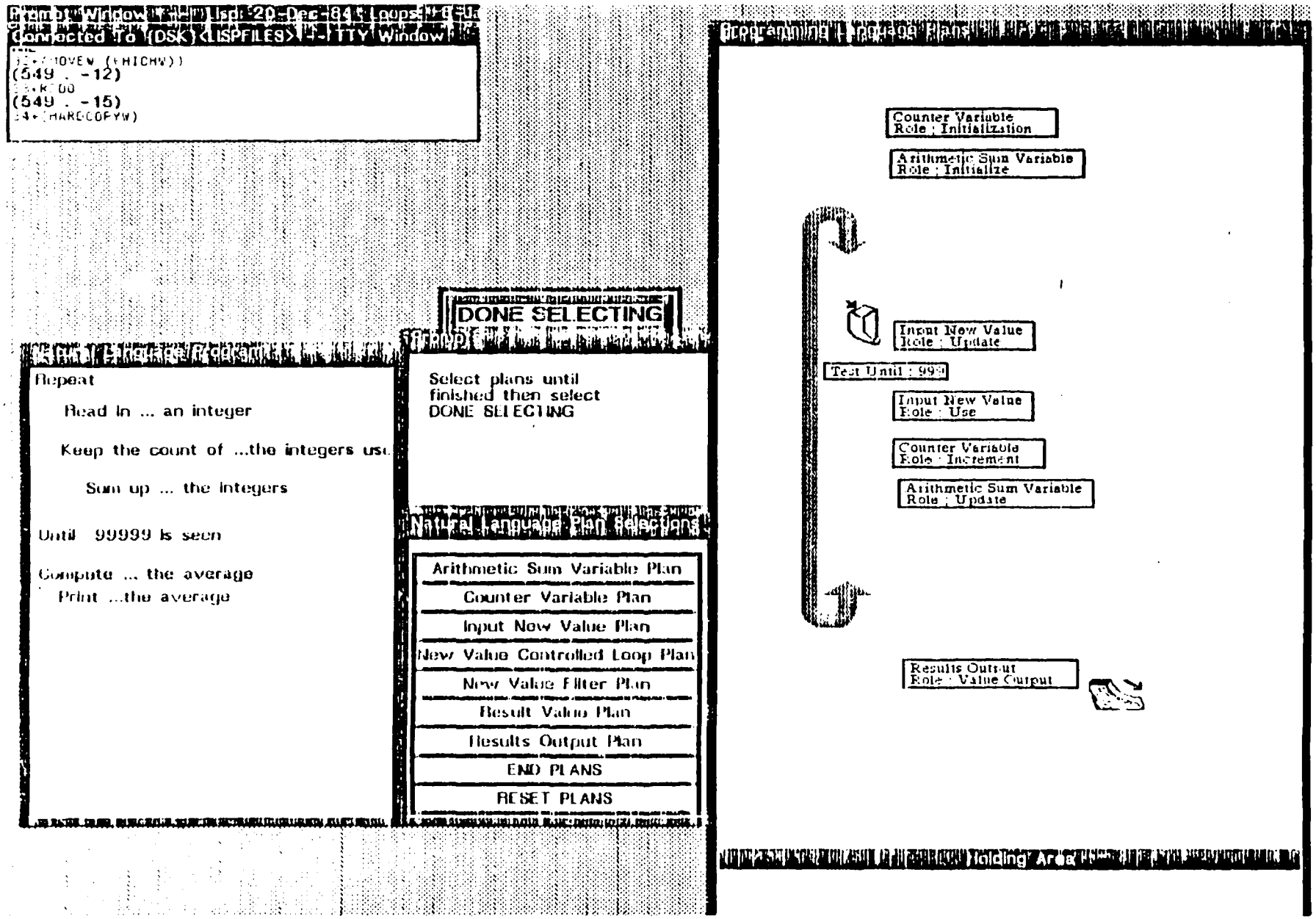

Connected To (DSK)<LISPFILES> - TTY Window
(549 . -12)
(549 . -15)
DONE SELECTING
Repeat
Read In ... an integer
Keep the count of ...the integers use
Sum up ... the integers
Until 99999 is seen
Compute ... the average
Print ...the average
Select plans until finished then select DONE SELECTING
Natural Language Plan Selections
Arithmetic Sum Variable Plan
Counter Variable Plan
Input New Value Plan
New Value Controlled Loop Plan
New Value Filter Plan
Result Value Plan
Results Output Plan
END PLANS
RESET PLANS
Counter Variable
Role : Initialization
Arithmetic Sum Variable
Role : Initialize
Input New Value
Role : Update
Test Until : 99999
Input New Value
Role : Use
Counter Variable
Role : Increment
Arithmetic Sum Variable
Role : Update
Results Output
Role : Value Output

FIGURE 10

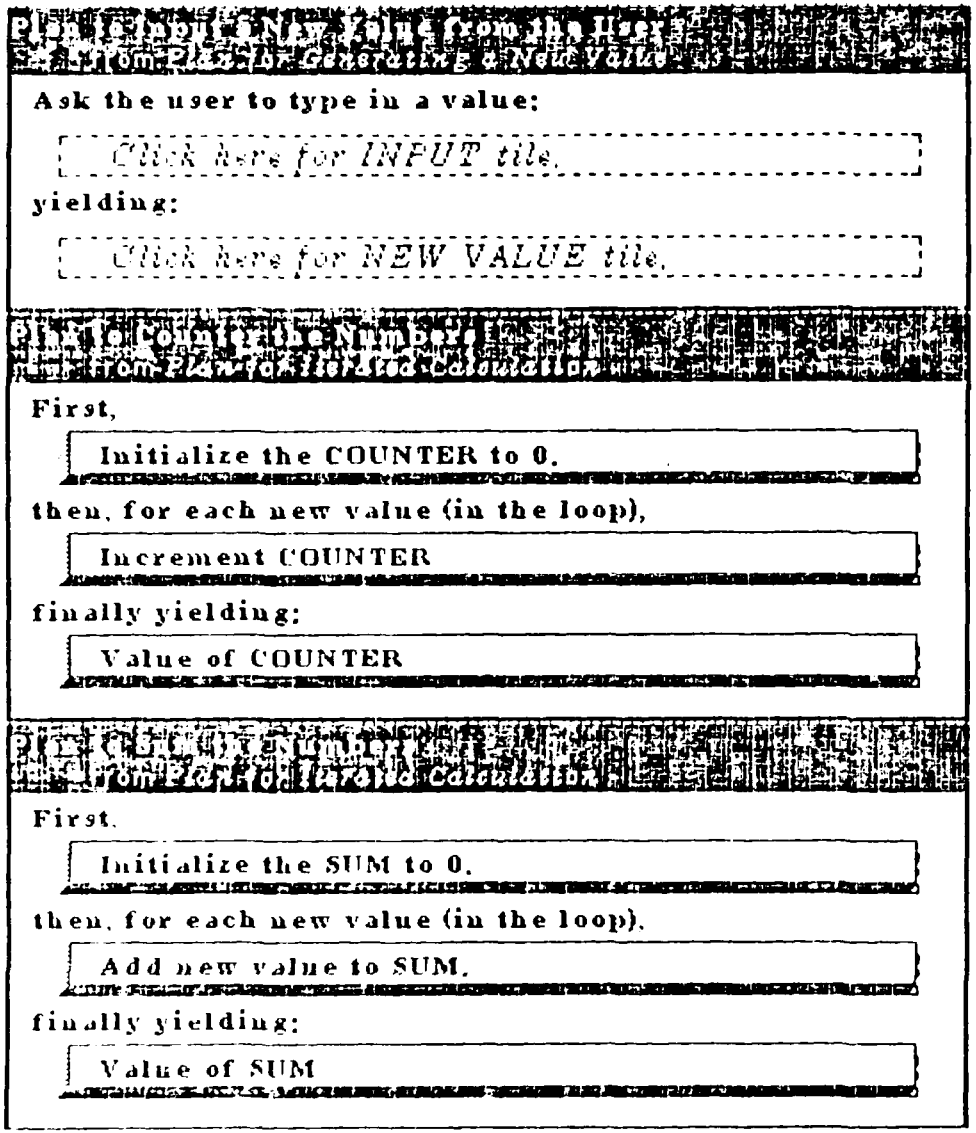

Ask the user to type in a value:
Click here for INPUT tile.
yielding:
Click here for NEW VALUE tile.
First,
Initialize the COUNTER to 0.
then, for each new value (in the loop),
Increment COUNTER
finally yielding:
Value of COUNTER
First,
Initialize the SUM to 0.
then, for each new value (in the loop),
Add new value to SUM.
finally yielding:
Value of SUM

FIGURE 11

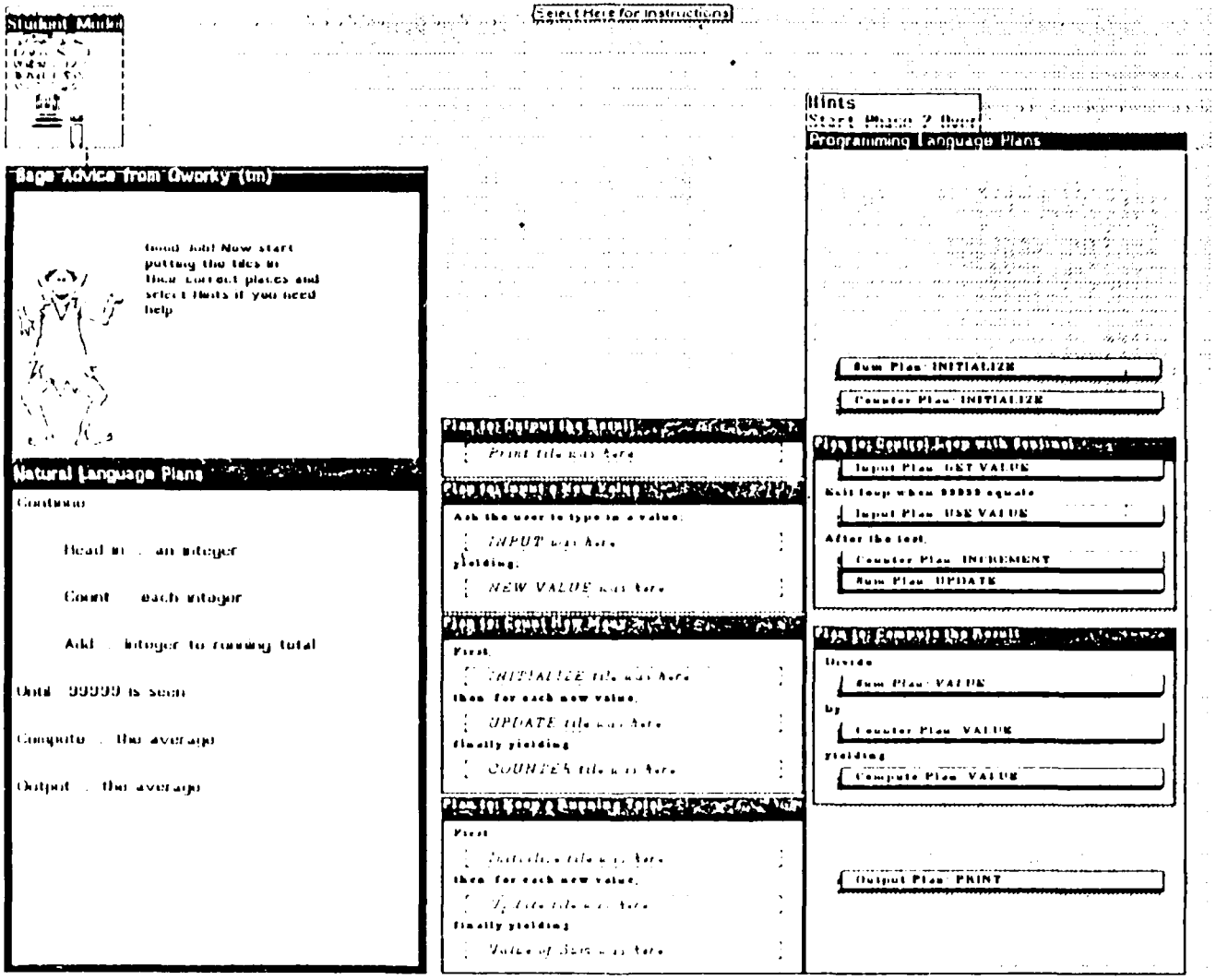

Select Here for Instructions
Student Model
Sage Advice from Qwerky (tm)
Good Job! Now start putting the tiles in their correct places and select Hints if you need help
Natural Language Plans
Continue
Read in an integer
Count each integer
Add integer to running total
Until 99999 is seen
Compute the average
Output the average
Hints
Programming Language Plans
Sum Plan: INITIALIZE
Counter Plan: INITIALIZE
Input Plan: GET VALUE
Exit loop when 99999 equals
Input Plan: USE VALUE
After the test,
Counter Plan: INCREMENT
Sum Plan: UPDATE
Divide
Sum Plan: VALUE
by
Counter Plan: VALUE
yielding
Compute Plan: VALUE
Output Plan: PRINT
Ask the user to type in a value,
yielding,
First,
then for each new value,
finally yielding
First
then for each new value,
finally yielding

FIGURE 12

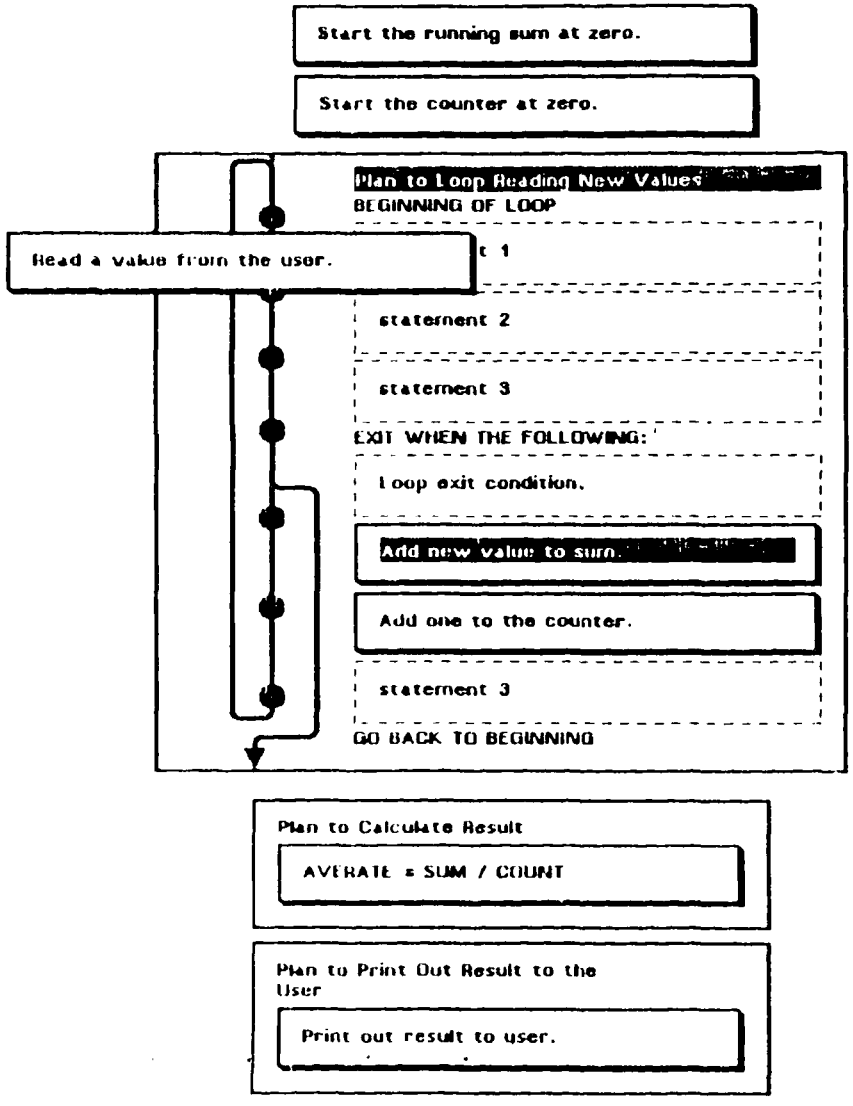

Start the running sum at zero.
Start the counter at zero.
Plan to Loop Reading New Values
BEGINNING OF LOOP
Read a value from the user.
t 1
statement 2
statement 3
EXIT WHEN THE FOLLOWING:
Loop exit condition.
Add new value to sum.
Add one to the counter.
statement 3
GO BACK TO BEGINNING
Plan to Calculate Result
AVERATE = SUM / COUNT
Plan to Print Out Result to the User
Print out result to user.

FIGURE 13

This approach also was too textual, giving up any advantage that might be gained from a more graphical representation. While the use of graphics in the previous version of was not particularly innovative, there at least was some suggestion of the dynamic flow of control.

Finally, this version did not account for the prospect of nested plans. How might the Compute the Result plan be placed inside the Control Loop with Sentinel plan, for instance?

4.3 Generation 3

The next version added more suggestive shape to the representation. In addition, it attempted to deal with the nesting issue. Figure 13 shows a loop with other plans following it. Preceding the loop plan are representations of initialization roles for two different variable plans. The basic representation is still, however, based on the notion of frames and slots.

Although this version adds some graphics, as in the original version the graphics are not particularly intuitive. In addition, they are still static. Up to this point in the development, however, it was not clear that an executable version of this phase was desireable. After lengthy consideration, it was decided to approach the phase with an iconic language that was executable.

4.4 Generation 4

This is the first entirely graphical representation of BridgeTalk. Each plan is represented by its own graphical object. The focus of this representation was on the merging and coordination of the plans, for instance as in the case of the actions to be performed as the body of a loop. In addition, this version was executable, allowing the user to see the flow of control as well as changes in data.

Figure 14 shows an object representing a generalized Loop plan and a Compute plan. In this case, the Loop plan currently contains a Counter plan and a Running Total plan. The Loop Test is on a slide mechanism, and the user must position the test with respect to other actions to be taken inside the loop. The Increment by and Update by roles of the plans embedded in the loop are also on slides. The position of the Initialize role of each of these plans is designed to indicate that they occur before the loop begins.

The Value roles hold the current value of each of the plans. The boxes containing these values are given unique shading to make them distinguishable throughout the program being created, as in the value implied by the Input plan given in the Loop plan. These values can be copied to other locations in the program.

In Figure 15 you can see how these mechanisms work in a typical example. In this example, the test condition of the loop has been positioned to occur before the Update

and Increment portions of the plans embedded in the loop. Also note that the value generated by the Input plan is the same value used by the Running Total plan in its Update by role (the shading of the two boxes is the same). Also, the Value role of the Running Total is used in the Compute plan, along with the Value from the Counter plan to produce the final Output of the program.

Note that an explicit flow control line has been attached between the Loop plan and the Compute plan, and between the Compute plan and the Output plan. Also, there is a flow line on the side of the Loop plan. These lines are animated as the program is executed to indicate the order of execution. In the example shown in Figure 15, three numbers were entered, producing a total of 120. The average was computed as 40.

While this version has the virtue of the suggestiveness of its icons, and it is dynamic in nature, there is still at least one major problem that it creates, namely the inconsistent semantics of its graphical constructs. For instance, control flow is shown in three different ways: as flow lines; as a top to bottom precedence (e.g. the fact that the Initialize roles of variable plans are executed before the loop because they appear on top of their respective plans, and the Loop Test occurring before the Update and Increment in the loop's body); and a left to right ordering (as evident in the loop body mechanism).

Another problem arises with the slide mechanisms used in some of the plans. What does the slide mean with regard to the Running Total and Counter plans? It may be somewhat clear as to why the slide in the Loop plan is necessary, but the only reason it is present in the other plans seems to be so that the Loop Test can be properly positioned. In addition, what happens if the Loop Test and the Update and/or Increment are positioned at the same level?

4.5 Generation 5

Part of the problems encountered by the version in generation 4 seemed to stem from the lack of an analogy for the figures used. There was no intuitive component that allowed users to recognize the meaning of the objects or their characteristics. In addition, as noted above, there was inconsistency in the flow of control.

Figure 16 shows a redesign of the graphical objects used to represent plans. The design was influenced by the Transformer type of toys, with pieces being keyed to fit particular places in other pieces. The flow of control was implied to be strictly left to right and top to bottom.

The design shown in Figure 16 turned out to be too "busy." Figure 17 shows a refinement of the graphics in order to make the objects a bit simpler. Figure 18 shows how some of these objects would fit together.

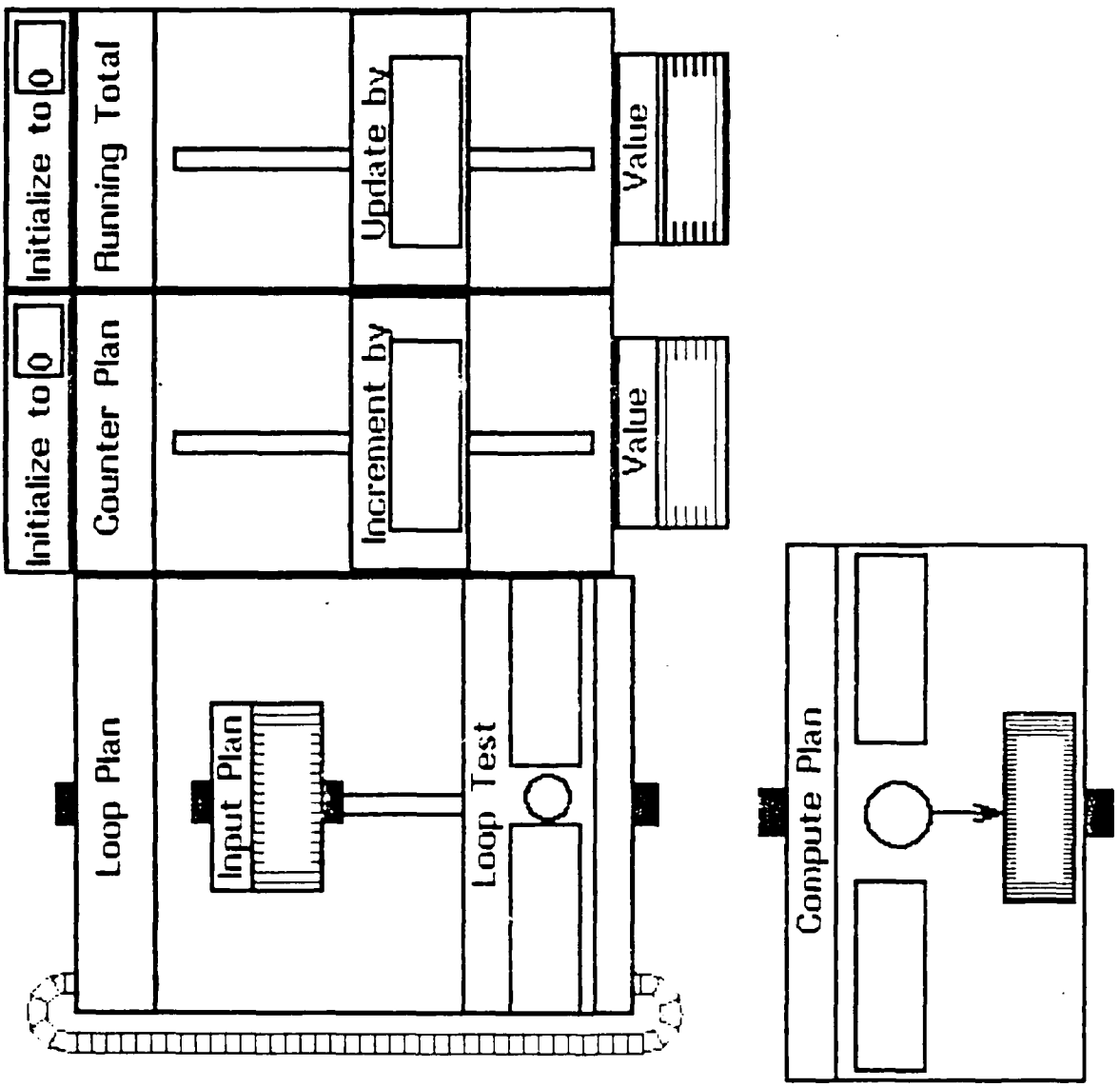

Initialize to 0
Running Total
Update by
Value
Initialize to 0
Counter Plan
Increment by
Value
Loop Plan
Input Plan
Loop Test
Compute Plan

FIGURE 14

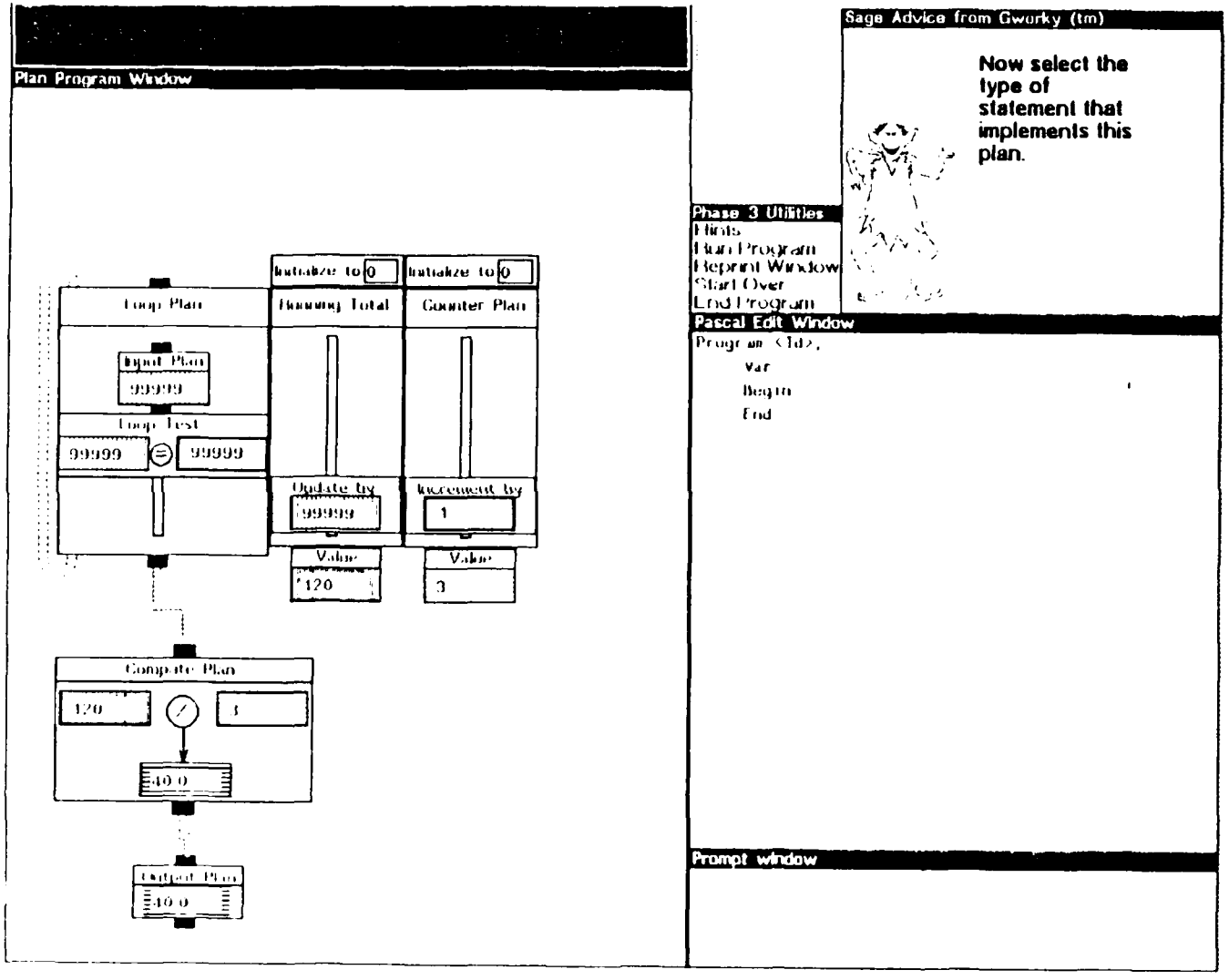
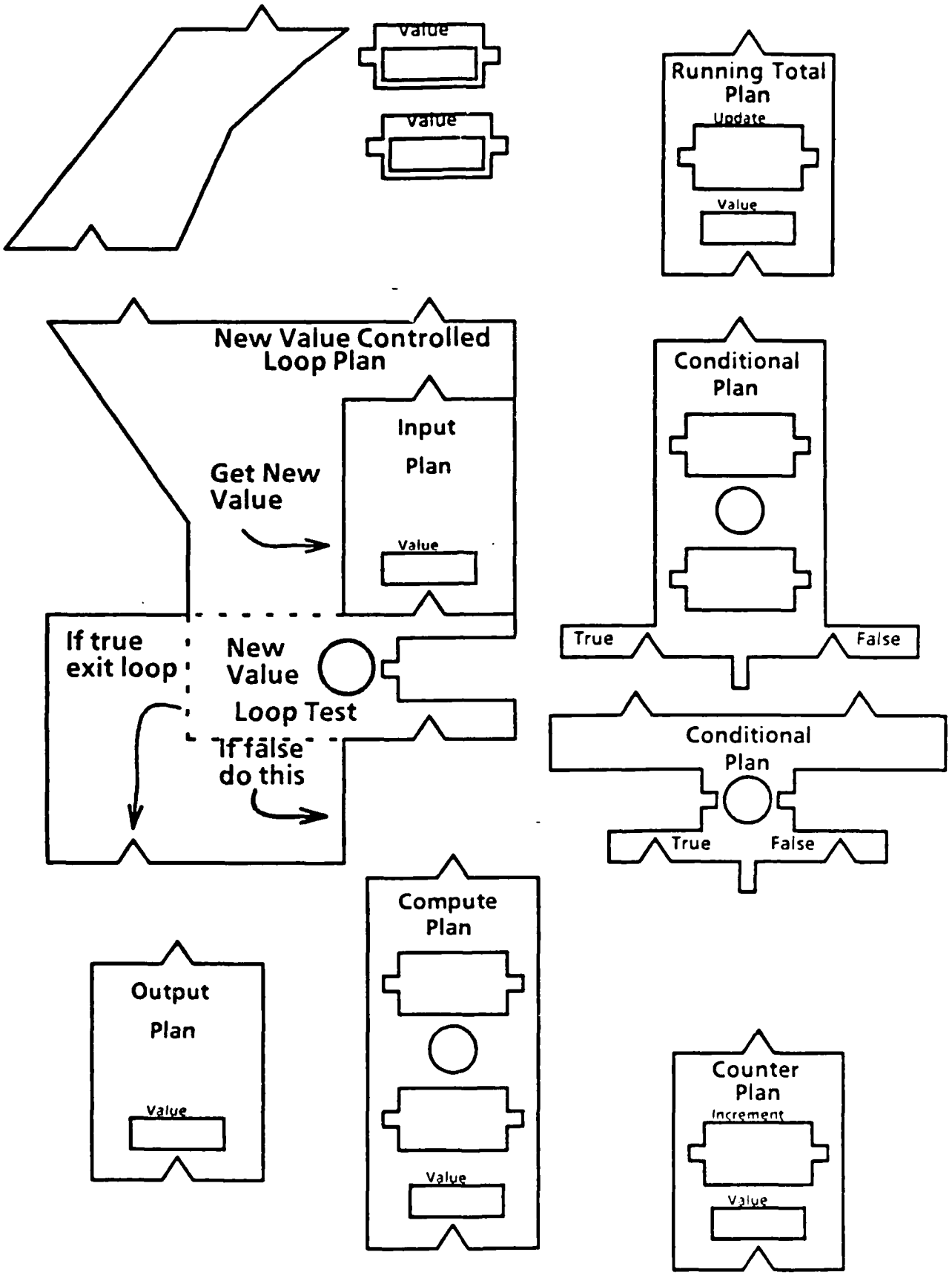
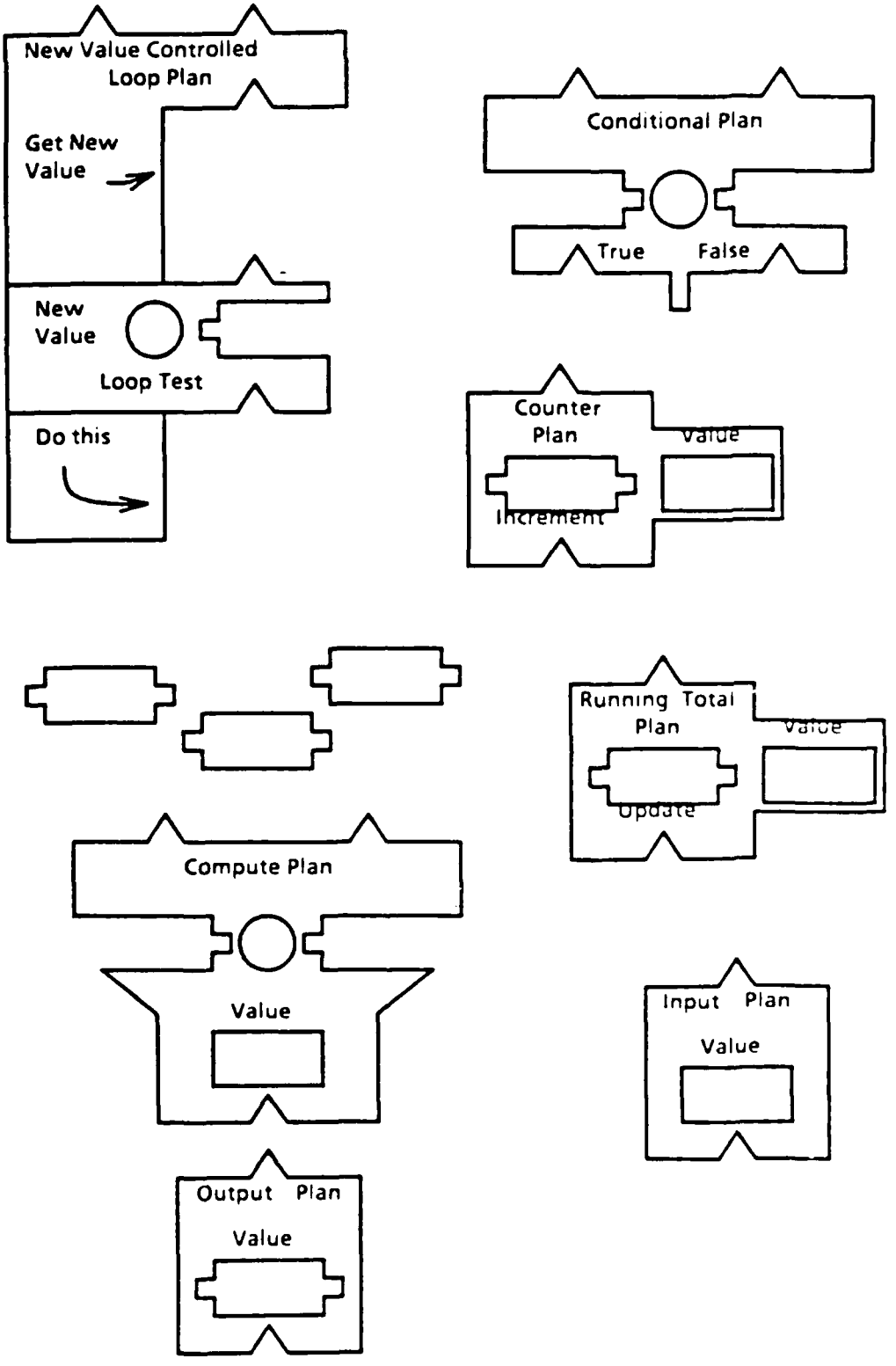
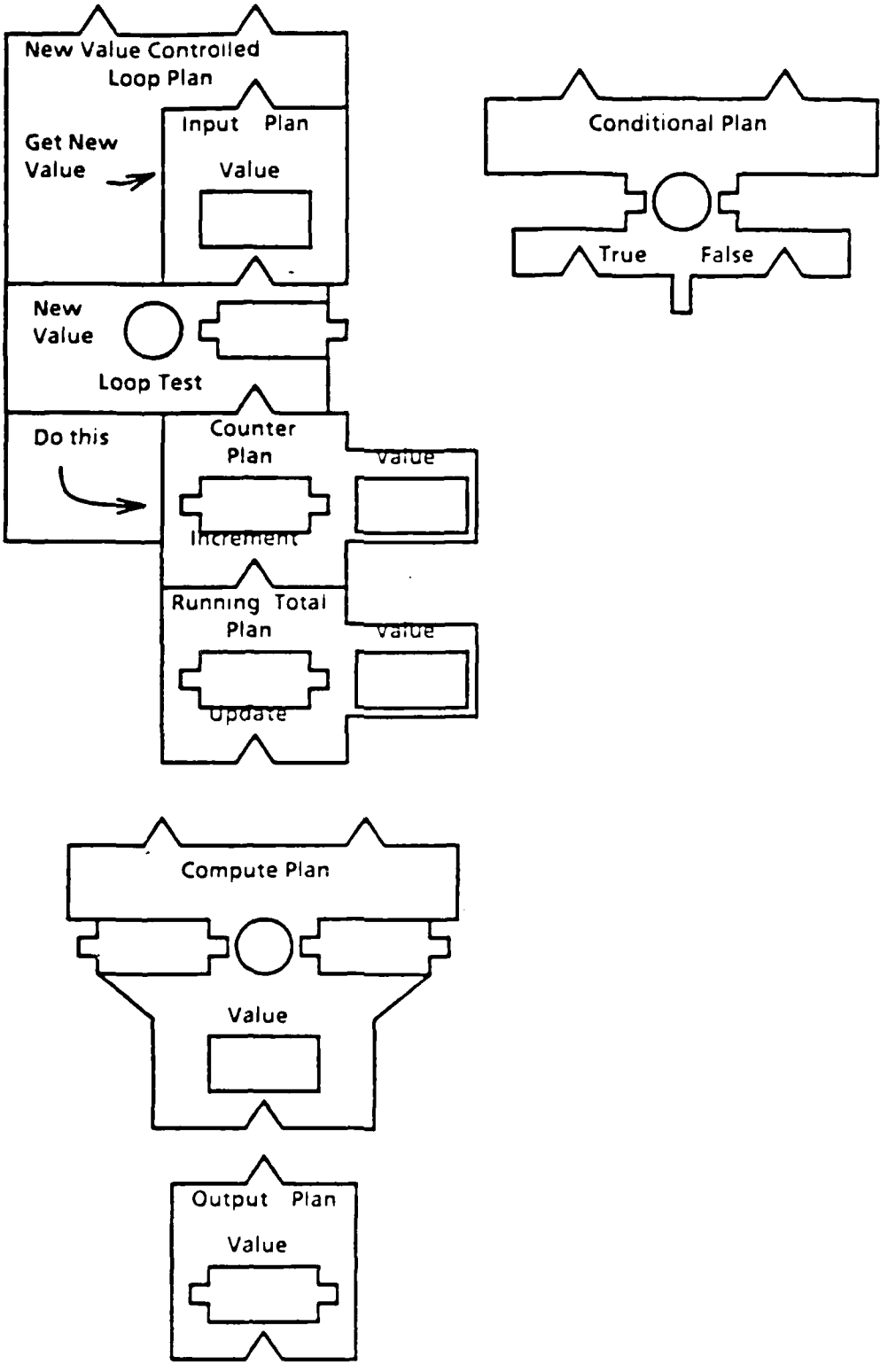
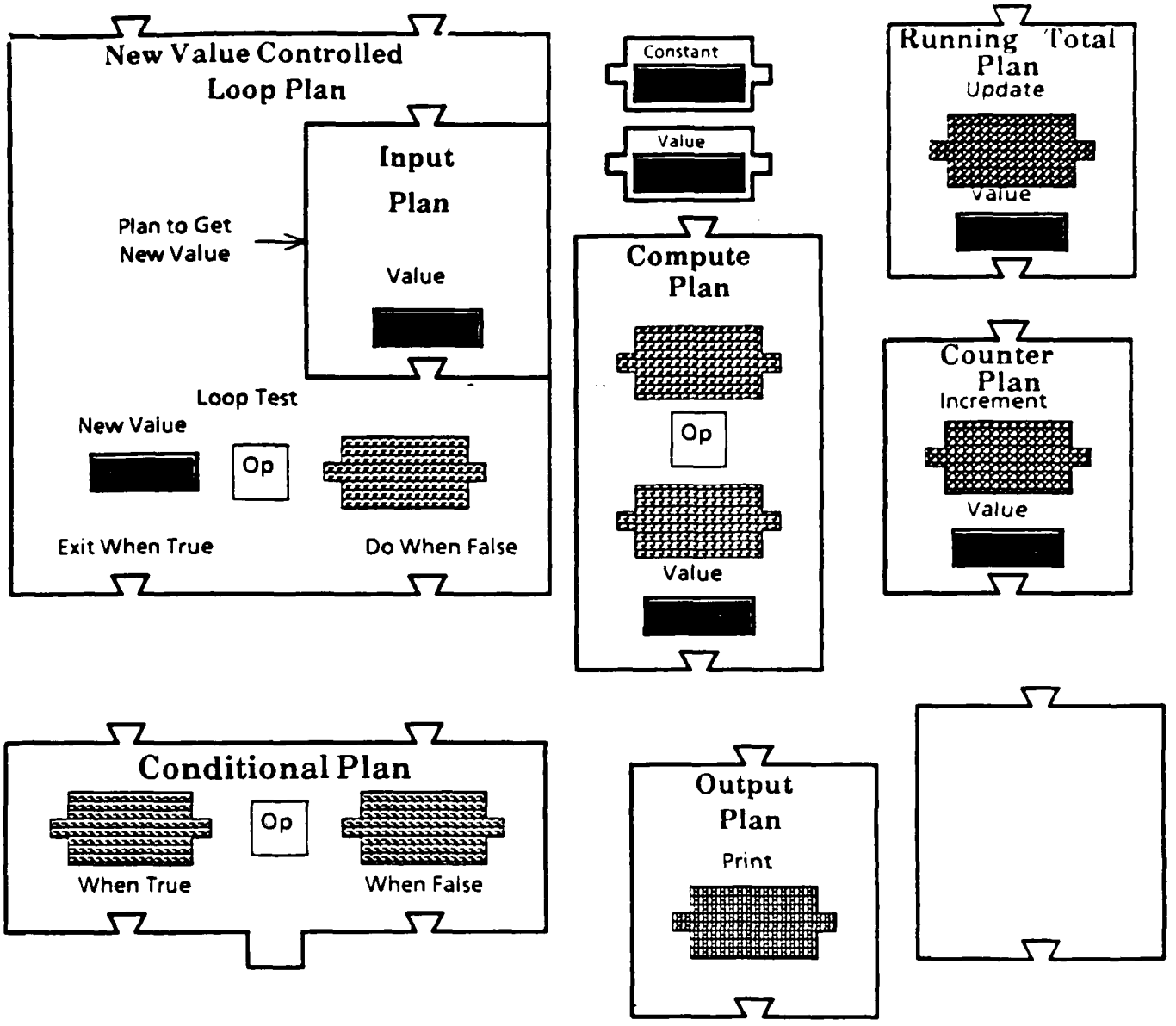

Plan Program Window
Initialize to 0
Initialize to 0
Loop Plan
Running Total
Counter Plan
Input Plan
99999
Loop Test
99999
=
99999
Update by
99999
Increment by
1
Value
120
Value
3
Compute Plan
120
/
3
40.0
Output Plan
40.0
Sage Advice from Gwurky (tm)
Now select the type of statement that implements this plan.
Phase 3 Utilities
Hints
Run Program
Repaint Window
Start Over
End Program
Pascal Edit Window
Program <Id>;
Var
Begin
End
Prompt window

FIGURE 15


Value
Value
Running Total Plan
Update
Value
New Value Controlled Loop Plan
Input Plan
Get New Value
Value
If true exit loop
New Value
Loop Test
If false do this
Conditional Plan
True
False
Conditional Plan
True
False
Compute Plan
Value
Output Plan
Value
Counter Plan
Increment
Value


New Value Controlled
Loop Plan
Get New
Value
New
Value
Loop Test
Do this
Conditional Plan
True
False
Counter
Plan
Value
Increment
Running Total
Plan
Value
Update
Compute Plan
Value
Input Plan
Value
Output Plan
Value


New Value Controlled
Loop Plan
Input Plan
Value
Get New
Value
New
Value
Loop Test
Do this
Counter
Plan
Value
Increment
Running Total
Plan
Value
Update
Conditional Plan
True
False
Compute Plan
Value
Output Plan
Value


New Value Controlled Loop Plan
Input Plan
Plan to Get New Value
Value
Loop Test
New Value
Op
Exit When True
Do When False
Constant
Value
Compute Plan
Op
Value
Running Total Plan
Update
Value
Counter Plan
Increment
Value
Conditional Plan
Op
When True
When False
Output Plan
Print

There were still some mechanical problems with this design, making the attachment of some of the objects difficult. In addition, it turned out that the left-right, top-bottom control flow was insufficient.

4.6 Generation 6

The current version of the visual language is shown in Figure 19. This design was influenced by jig-saw puzzle pieces. The mechanics of this version work out better than the previous version. In addition, control flow is more explicit, following the keyed tabs on each object top-down. Data objects are rectangular shapes with tabs on their left and right sides. Originally, these data objects were placed over a recepticle that contained a "velco" substance for sticking them to. This was later changed to being simply holes that were filled with the data objects.

Figure 20 shows a solution to an averaging problem. During the running of the visual solution, data flow is explicitly provided by having the values actually move from one location to another. For instance, in this example, when a value is entered as an input, the value would automatically be carried by a moving box to the New Value location in the New Value Controlled Loop plan, and on down to the Update location in the Running Total plan. Likewise, the Value from the Running Total plan and from the Counter plan would be carried to the appropriate locations in the Compute plan. The Value in the Compute plan would be carried down to the Output plan.

There are still problems to be overcome in this current version of the visual language. For instance, the issue of nesting plans has still not been fully addressed. In addition, there is still some concern that the shapes of the plans are not adequate in their ability to evoke understanding by the user.

Despite the difficulties with the current version, what is clear at this point is that the use of a visual language is effective.

5. Future Steps

There are several issues still to be resolved with BridgeTalk. These issues fall into the general areas of visual appearence, implementation, and usage. We take up each topic separately.

The most problematic issue with the visual appearence of BridgeTalk is the difficulty of expressing nesting. In particular, the current version can visually express only one level of nesting, and that one level with only a fixed number of contained elements. The problem is that there must be only a small finite amount of space to put a contained element if the parent icon is visually containing them. Inevitably, we will need to go to some sort of grow shrink scheme for the contained elements. We have

resisted this because it is inherently disturbing and confusing unless the grow/shrink transistion is smooth and obviously reversable. Most visual languages address this problem by using small icon elements that expand to windows. This solution is unacceptable due to our experience suggesting the importance of icon shapes suggesting the icon usage.

The BridgeTalk implementation is still incomplete. In particular, we still have not completed the browsers that allow a user to navigate the structures showing plan inheritance and implementation, as pictured in Figures 8 and 9. We will have a challenge in distinguishing this representation from the specific plan icons used at the surface level. On the otherhand, we feel these structures are a crucial tool for teaching students to understand the management of abstraction.

The final area of future work suggested by this work is the most complex: how is this tool and approach best used? In our attempt to tease apart the expression of intentions to a computer, we have become increasingly suspicious of the glib discussion of "top-down design" found in most programming textbooks. In particular, the design of BridgeTalk shows that programming design involves many different mappings, including: informal to formal, declarative to procedural, goals to plans and processes, natural language to Pascal, linear structure to tree structure, and weakly constrained to strongly constrained. We believe that programming texts do their students a disservice by presenting a design model that at best ignores the differences between novices and experts, and at worst is completly unrelated to actual programming practice.

6. Conclusions

We have reported on BridgeTalk, a new approach to visual languages for novice programmers. BridgeTalk is based on actual data about how novices learn to program. It allows novices to program with programming plans, and focuses novices on the interactions between plans, not on the implementation details for a particular plan. Beginning with plans as a basis for a novice programming language, we were forced to develop a programming formalism that could deal with multiple levels of detail, merged plan implementations, and interrelationships between plans. Finally, we used six "design, implement, test, and redesign" cycles to develop a specific visual representation for the language.

A key contribution of this work is the systematic support for a plan-like view of programming. We feel that such support is essential if programming languages are to allow a programmer, novice or expert, to work with programming constructs that reflect actual world semantics. Unfortunately, "high level programming language" has

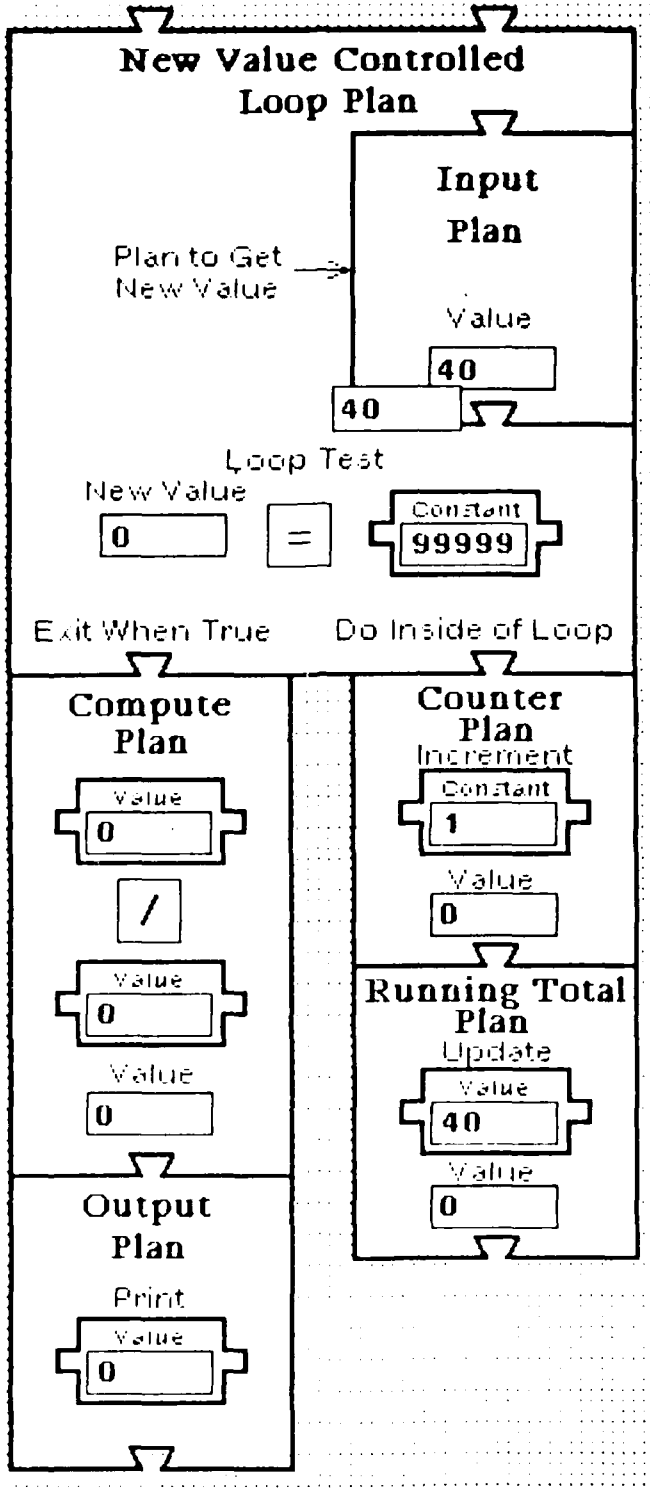

New Value Controlled Loop Plan
Input Plan
Plan to Get New Value
Value
40
40
Loop Test
New Value
0
=
Constant
99999
Exit When True
Do Inside of Loop
Compute Plan
Value
0
/
Value
0
Value
0
Counter Plan
Increment
Constant
1
Value
0
Running Total Plan
Update
Value
40
Value
0
Output Plan
Print
Value
0

FIGURE 20

come to mean more abstract data structures and more formal semantics. We advocate, and have demonstrated the feasability of, "high level languages" that reflect the semantics of real-world objects.

Consistant with a language that makes a richer connection to the semantics of real-world objects, our language provides novices with a pathway from experience in the real-world, to real-world plans, to standard programming language constructs. Current software systems force a user to either use the system as is, or become a programmer. BridgeTalk illustrates an approach that provides a smoother transition with many intermediate steps along the way. That is, programming strictly with plans simplifies programming by allowing users to draw on their knowledge of real-world situations, but limits users to those plans provided in the system. Because the internals of the plans can be examined, modified, and specialized, users can extend the power of their system beyond the predefined plans.

One of the most important lessons of this work is the criticality of empiricial work in the design of visual languages. Without the extensive empirical work documented in this report, our language would be much poorer. Furthermore, the language would more than likely be unsuitable for use with novices. There is no reason to believe that languages designed based on purely formal and intuitive arguments will be usable. Until a suitable theory of visual interface design emerges, we feel that computer scientists must submit their visual languages to the test of real users under realistic circumstances. Not only will this produce happier languages, we believe, but also better science.

7. References

Bonar, J., Cunningham, R., Beatty, P., & Riggs, P. [1987]. Bridge: Intelligent Tutoring With Intermediate Representations. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260

Bonar, J., and Soloway, E. [1985]. Pre-programming Knowledge: A Major Source of Misconceptions in Novice Programmers. Human-Computer Interaction, 1.

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

Cunningham, R., Corbett, J., & Bonar, J. [1987] The Chips Technical Report. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260

Liffick, B. [1987]. A Visual Programming Environment for Novices. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260

Rich, C. [1981]. A Formal Representation for Plans in the Programmer's Apprentice. *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981. Also in *Artificial Intelligence and Software Engineering*, Rich, C. and Waters, R.C (ed.). Morgan Kaufmann Publishing, 1986

Shrobe, H. [1979]. Dependency Directed Reasoning for Complex Program Understanding. PhD Thesis. MIT/AI/TR-503, April 1979

Soloway, E. Bonar, J., & Ehrlich, K. [1983]. Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM, 26*, November.

Soloway, E., & Ehrlich, K. [1984]. Emperical Studies of Programming Knowledge. *IEEE Transactions of Software Engineering, SE-10*

Spohrer, J., Soloway, E., & Pope E. [1985]. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction, 1*

Waters, R.C. [1978]. Automatic Analysis of the Logical Structure of Programs. PhD Thesis. MIT/AI/TR-492, December 1978

Waters, R.C. [1985]. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions of Software Engineering, SE-11*

Winograd, T. [1979]. Beyond Programming Languages. *Communications of the ACM*, July 1979. Also in *Interactive Programming Environments*, Barstow, D. et al (ed.), McGraw-Hill, 1984

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Phillip L. Ackerman
University of Minnesota
Department of Psychology
Minneapolis, MN 55455

Dr. Beth Adelson
Department of Computer Science
Tufts University
Medford, MA 02155

Air Force Human Resources Lab
AFHRL/MPD
Brooks AFB, TX 78235

AFOSR,
Life Sciences Directorate
Bolling Air Force Base
Washington, DC 20332

Technical Director,
Army Human Engineering Lab
Aberdeen Proving Ground
MD 21005

Dr. Robert Ahlers
Code N711
Human Factors Laboratory
Naval Training Systems Center
Orlando, FL 32813

Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. John Allen
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Dr. James Anderson
Brown University
Center for Neural Science
Providence, RI 02912

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Nancy S. Anderson
Department of Psychology
University of Maryland
College Park, MD 20742

Technical Director, ARI
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

Dr. Patricia Baggett
University of Colorado
Department of Psychology
Box 345
Boulder, CO 80309

Dr. Eva L. Baker
UCLA Center for the Study
of Evaluation
145 Moore Hall
University of California
Los Angeles, CA 90024

Dr. Meryl S. Baker
Navy Personnel R&D Center
San Diego, CA 92152-6800

prof. dott. Bruno G. Bara
Unita di ricerca di
intelligenza artificiale
Universita di Milano
20122 Milano - via F. Sforza 23
ITALY

Dr. William M. Bart
University of Minnesota
Dept. of Educ. Psychology
330 Burton Hall
178 Pillsbury Dr., S.E.
Minneapolis, MN 55455

Dr. Jackson Beatty
Department of Psychology
University of California
Los Angeles, CA 90024

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. John Black
Teachers College
Columbia University
525 West 121st Street
New York, NY 10027

Dr. Jeff Bonar
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Gordon H. Bower
Department of Psychology
Stanford University
Stanford, CA 94306

Dr. Robert Breaux
Code N-095R
Naval Training Systems Center
Orlando, FL 32813

Commanding Officer
CAPT Lorin W. Brown
NROTC Unit
Illinois Institute of Technology
3300 S. Federal Street
Chicago, IL 60616-3793

Dr. John S. Brown
XEROX Palo Alto Research
Center
3333 Coyote Road
Palo Alto, CA 94304

Maj. Hugh Burns
AFHRL/IDE
Lowry AFB, CO 80230-5000

Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. Pat Carpenter
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

LCDR Robert Carter
Office of the Chief
of Naval Operations
OP-01B
Pentagon
Washington, DC 20350-2000

Chair, Department of
Psychology
College of Arts and Sciences
Catholic University of
America
Washington, DC 20064

Dr. Michelene Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Dr. L. J. Chmura
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Mr. Raymond E. Christal
AFHRL/MOE
Brooks AFB, TX 78235

Assistant Chief of Staff
for Research, Development,
Test, and Evaluation
Naval Education and
Training Command (N-5)
NAS Pensacola, FL 32508

Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

Dr. Stanley Collyer
Office of Naval Technology
Code 222
800 N. Quincy Street
Arlington, VA 22217-5000

Brian Dallman
3400 TTW/TTGXS
Lowry AFB, CO 80230-5000

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Diane Damos
Arizona State University
Department of Psychology
Tempe, AZ 85287

Dr. Denise Dellarosa
Department of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

Dr. R. K. Dismukes
Associate Director for Life Sciences
AFOSR
Bolling AFB
Washington, DC 20332

Dr. Stephanie Doan
Code 6021
Naval Air Development Center
Warminster, PA 18974-5000

Dr. Emanuel Donchin
University of Illinois
Department of Psychology
Champaign, IL 61820

Defense Technical
Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC
(12 Copies)

Dr. Susan Embretson
University of Kansas
Psychology Department
426 Fraser
Lawrence, KS 66045

Dr. Randy Engle
Department of Psychology
University of South Carolina
Columbia, SC 29208

Dr. William Epstein
University of Wisconsin
W. J. Brogden Psychology Bldg.
1202 W. Johnson Street
Madison, WI 53706

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, MD 20014

Dr. K. Anders Ericsson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Martha Farah
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. Beatrice J. Farr
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Marshall J. Farr
Farr-Sight Co.
2520 North Vernon Street
Arlington, VA 22207

Dr. Paul Feltovich
Southern Illinois University
School of Medicine
Medical Education Department
P.O. Box 3926
Springfield, IL 62708

Dr. Craig I. Fields
ARPA
1400 Wilson Blvd.
Arlington, VA 22209

J. D. Fletcher
9931 Corsica Street
Vienna VA 22180

Dr. Kenneth D. Forbus
University of Illinois
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Alfred R. Fregly
AFOSR/NL
Bolling AFB, DC 20332

Dr. Michael Friendly
Psychology Department
York University
Toronto ONT
CANADA M3J 1P3

Julie A. Gadsden
Information Technology
Applications Division
Admiralty Research Establishment
Portsdown, Portsmouth PO6 4AA
UNITED KINGDOM

Dr. Michael Genesereth
Stanford University
Computer Science Department
Stanford, CA 94305

Dr. Dedre Gentner
University of Illinois
Department of Psychology
603 E. Daniel St.
Champaign, IL 61820

Dr. Lee Giles
AFOSR
Bolling AFB
Washington, DC 20332

Dr. Robert Glaser
Learning Research
& Development Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Dr. Marvin D. Glock
13 Stone Hall
Cornell University
Ithaca, NY 14853

Dr. Sam Glucksberg
Department of Psychology
Princeton University
Princeton, NJ 08540

Dr. Daniel Gopher
Industrial Engineering
& Management
TECHNION
Haifa 32000
ISRAEL

Dr. Sherrie Gott
AFHRL/MODJ
Brooks AFB, TX 78235

Dr. T. Govindaraj
Georgia Institute of Technology
School of Industrial & Systems
Engineering
Atlanta, GA 30332

Dr. Richard H. Granger
Department of Computer Science
University of California, Irvine
Irvine, CA 92717

Dr. James G. Greeno
University of California
Berkeley, CA 94720

Dr. Henry M. Halff
Halff Resources, Inc.
4918 33rd Road, North
Arlington, VA 22207

Dr. Bruce Hamill
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20707

Dr. John M. Hammer
Center for Man-Machine
Systems Research
Georgia Institute of Technology
Atlanta, GA 30332

Dr. Ray Hannapel
Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

Dr. Harold Hawkins
Office of Naval Research
Code 1142CS
800 N. Quincy Street
Arlington, VA 22217-5000

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

Dr. Joan I. Heller
505 Haddon Road
Oakland, CA 94606

Dr. Geoffrey Hinton
Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213

Dr. James D. Hollan
MCC,
Human Interface Program
3500 West Balcones Center Dr.
Austin, TX 78759

Dr. John Holland
University of Michigan
2313 East Engineering
Ann Arbor, MI 48109

Dr. Melissa Holland
Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Robert W. Holt
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Ms. Julia S. Hough
Lawrence Erlbaum Associates
6012 Greene Street
Philadelphia, PA 19144

Dr. James Howard
Dept. of Psychology
Human Performance Laboratory
Catholic University of
America
Washington, DC 20064

Dr. Earl Hunt
Department of Psychology
University of Washington
Seattle, WA 98105

Dr. Ed Hutchins
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Janet Jackson
Rijksuniversiteit Groningen
Biologisch Centrum, Vleugel D
Kerklaan 30, 9751 NN Haren (Gn.)
NETHERLANDS

Dr. R. J. K. Jacob
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Dr. Zachary Jacobson
Bureau of Management Consulting
365 Laurier Avenue West
Ottawa, Ontario K1A 0S5
CANADA

Pharm.-Chim. en Chef Jean Jacq
Division de Psychologie
Centre de Recherches du
Service de Sante des Armees
108 Boulevard Pinel
69272 Lyon Cedex 03, FRANCE

Dr. Robert Jannarone
Department of Psychology
University of South Carolina
Columbia, SC 29208

Dr. Claude Janvier
Directeur, CIRADE
Universite' du Quebec a Montreal
P.O. Box 8888, St. "A"
Montreal, Quebec H3C 3P8
CANADA

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

COL Dennis W. Jarvi
Commander
AFHRL
Brooks AFB, TX 78235-5601

Dr. Robin Jeffries
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971

Dr. Douglas H. Jones
Thatcher Jones Associates
P.O. Box 6640
10 Trafalgar Court
Lawrenceville, NJ 08648

Dr. Marcel Just
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Daniel Kahneman
Department of Psychology
University of California
Berkeley, CA 94720

Dr. Milton S. Katz
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Steven W. Keele
Department of Psychology
University of Oregon
Eugene, OR 97403

Dr. Wendy Kellogg
IBM T. J. Watson Research Ctr.
P.O. Box 218
Yorktown Heights, NY 10598

Dr. David Kieras
University of Michigan
Technical Communication
College of Engineering
1223 E. Engineering Building
Ann Arbor, MI 48109

Dr. Walter Kintsch
Department of Psychology
University of Colorado
Campus Box 345
Boulder, CO 80302

Dr. David Klahr
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Mr. Al Kleider
Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709-2211

Dr. Ronald Knoll
Bell Laboratories
Murray Hill, NJ 07974

Dr. Stephen Kosslyn
Harvard University
1236 William James Hall
33 Kirkland St.
Cambridge, MA 02138

Dr. Kenneth Kotovsky
Department of Psychology
Community College of
Allegheny County
800 Allegheny Avenue
Pittsburgh, PA 15233

Dr. David H. Krantz
2 Washington Square Village
Apt. # 15J
New York, NY 10012

Dr. Patrick Kyllonen
325 Aderhold
Department of Educational
Psychology
University of Georgia
Athens, GA 30602

Dr. David R. Lambert
Naval Ocean Systems Center
Code 441T
271 Catalina Boulevard
San Diego, CA 92152-6800

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Jill Larkin
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. R. W. Lawler
ARI 6 S 10
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

Dr. Alan M. Lesgold
Learning Research and
Development Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Alan Leshner
Deputy Division Director
Behavioral and Neural Sciences
National Science Foundation
1800 G Street
Washington, DC 20550

Dr. Jim Levin
Department of
Educational Psychology
210 Education Building
1310 South Sixth Street
Champaign, IL 61820-6990

Dr. John Levine
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Clayton Lewis
University of Colorado
Department of Computer Science
Campus Box 430
Boulder, CO 80309

Matt Lewis
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Library,
Naval War College
Newport, RI 02940

Library,
Naval Training Systems
Center
Orlando, FL 32813

Science and Technology Division,
Library of Congress
Washington, DC 20540

Dr. Jane Malin
Mail Code SR 111
NASA Johnson Space Center
Houston, TX 77058

Dr. Sandra P. Marshall
Dept. of Psychology
San Diego State University
San Diego, CA 92182

Dr. Humberto Maturana
University of Chile
Santiago
CHILE

Dr. Richard E. Mayer
Department of Psychology
University of California
Santa Barbara, CA 93106

Dr. James McBride
Psychological Corporation
c/o Harcourt, Brace,
Javanovich Inc.
1250 West 6th Street
San Diego, CA 92101

Dr. James L. McGaugh
Center for the Neurobiology
of Learning and Memory
University of California, Irvine
Irvine, CA 92717

Dr. Gail McKoon
CAS/Psychology
Northwestern University
1859 Sheridan Road
Kresge #230
Evanston, IL 60201

Dr. Joe McLachlan
Navy Personnel R&D Center
San Diego, CA 92152-6800

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. James S. McMichael
Navy Personnel Research
and Development Center
Code 05
San Diego, CA 92152

Dr. Barbara Means
Human Resources
Research Organization
1100 South Washington
Alexandria, VA 22314

Dr. Douglas L. Medin
Department of Psychology
University of Illinois
603 E. Daniel Street
Champaign, IL 61820

Dr. George A. Miller
Department of Psychology
Green Hall
Princeton University
Princeton, NJ 08540

Dr. Andrew R. Molnar
Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

Dr. William Montague
NPRDC Code 13
San Diego, CA 92152-6800

Dr. Nancy Morris
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Randy Mumaw
Program Manager
Training Research Division
HumRRO
1100 S. Washington
Alexandria, VA 22314

Dr. Allen Munro
Behavioral Technology
Laboratories - USC
1845 S. Elena Ave., 4th Floor
Redondo Beach, CA 90277

Chair, Department of
Computer Science
U.S. Naval Academy
Annapolis, MD 21402

Chair, Department of
Systems Engineering
U.S. Naval Academy
Annapolis, MD 21402

Technical Director,
Navy Health Research Center
P.O. Box 85122
San Diego, CA 92138

Dr. Allen Newell
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. Mary Jo Nissen
University of Minnesota
N218 Elliott Hall
Minneapolis, MN 55455

Dr. A. F. Norcio
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Dr. Donald A. Norman
Institute for Cognitive
Science C-015
University of California, San Diego
La Jolla, California 92093

Deputy Technical Director,
NPRDC Code 01A
San Diego, CA 92152-6800

Director, Training Laboratory,
NPRDC (Code 05)
San Diego, CA 92152-6800

Director, Manpower and Personnel
Laboratory,
NPRDC (Code 06)
San Diego, CA 92152-6800

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Director, Human Factors
& Organizational Systems Lab,
NPRDC (Code 07)
San Diego, CA 92152-6800

Fleet Support Office,
NPRDC (Code 301)
San Diego, CA 92152-6800

Library, NPRDC
Code P201L
San Diego, CA 92152-6800

Technical Director,
Navy Personnel R&D Center
San Diego, CA 92152-6800

Commanding Officer,
Naval Research Laboratory
Code 2627
Washington, DC 20390

Dr. Harold F. O'Neil, Jr.
School of Education - WPH 801
Department of Educational
Psychology & Technology
University of Southern California
Los Angeles, CA 90089-0031

Dr. Michael Oberlin
Naval Training Systems Center
Code 711
Orlando, FL 32813-7100

Dr. Stellan Ohlsson
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Office of Naval Research,
Code 1142BI
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142
800 N. Quincy St.
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142PS
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142CS
800 N. Quincy Street
Arlington, VA 22217-5000
(6 Copies)

Psychologist,
Office of Naval Research
Branch Office, London
Box 39
FPO New York, NY 09510

Special Assistant for Marine
Corps Matters,
ONR Code 00MC
800 N. Quincy St.
Arlington, VA 22217-5000

Psychologist,
Office of Naval Research
Liaison Office, Far East
APO San Francisco, CA 96503

Dr. Judith Orasanu
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Douglas Pearse
DCIEM
Box 2000
Downsview, Ontario
CANADA

Dr. James W. Pellegrino
University of California,
Santa Barbara
Department of Psychology
Santa Barbara, CA 93106

Dr. Virginia E. Pendergrass
Code 711
Naval Training Systems Center
Orlando, FL 32813-7100

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Nancy Pennington
University of Chicago
Graduate School of Business
1101 E. 58th St.
Chicago, IL 60637

Military Assistant for Training and
Personnel Technology,
OUSD (R & E)
Room 3D129, The Pentagon
Washington, DC 20301-3080

Dr. Steven Pinker
Department of Psychology
E10-018
M.I.T.
Cambridge, MA 02139

Dr. Martha Polson
Department of Psychology
Campus Box 346
University of Colorado
Boulder, CO 80309

Dr. Peter Polson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Michael I. Posner
Department of Neurology
Washington University
Medical School
St. Louis, MO 63110

Dr. Mary C. Potter
Department of Psychology
MIT (E-10-032)
Cambridge, MA 02139

Dr. Paul S. Rau
Code U-32
Naval Surface Weapons Center
White Oak Laboratory
Silver Spring, MD 20903

Dr. Lynne Reder
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. James A. Reggia
University of Maryland
School of Medicine
Department of Neurology
22 South Greene Street
Baltimore, MD 21201

Dr. Wesley Regian
AFHRL/MOD
Brooks AFB, TX 78235

Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

Dr. Gil Ricard
Mail Stop C04-14
Grumman Aerospace Corp.
Bethpage, NY 11714

Dr. Linda G. Roberts
Science, Education, and
Transportation Program
Office of Technology Assessment
Congress of the United States
Washington, DC 20510

Dr. Paul R. Rosenbaum
Educational Testing Service
Princeton, NJ 08541

Dr. William B. Rouse
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. David Rumelhart
Center for Human
Information Processing
Univ. of California
La Jolla, CA 92093

Dr. Walter Schneider
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Miriam Schustack
Code 51
Navy Personnel R & D Center
San Diego, CA 92152-6800

Dr. Marc Sebrechts
Department of Psychology
Wesleyan University
Middletown, CT 06475

Dr. Colleen M. Seifert
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Ben Shneiderman
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Dr. Robert S. Siegler
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Herbert A. Simon
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

LTCOL Robert Simpson
Defense Advanced Research
Projects Administration
1400 Wilson Blvd.
Arlington, VA 22209

Dr. H. Wallace Sinaiko
Manpower Research
and Advisory Services
Smithsonian Institution
801 North Pitt Street
Alexandria, VA 22314

Dr. Richard E. Snow
Department of Psychology
Stanford University
Stanford, CA 94306

Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Kathryn T. Spoehr
Brown University
Department of Psychology
Providence, RI 02912

Dr. James J. Staszewski
Research Associate
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Robert Sternberg
Department of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

Dr. Kurt Steuck
AFHRL/MOD
Brooks AFB
San Antonio TX 78235

Dr. Paul J. Sticha
Senior Staff Scientist
Training Research Division
HumRRO
1100 S. Washington
Alexandria, VA 22314

Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

Dr. Kikumi Tatsuoka
CERL
252 Engineering Research
Laboratory
Urbana, IL 61801

Dr. Perry W. Thorndyke
FMC Corporation
Central Engineering Labs
1185 Coleman Avenue, Box 580
Santa Clara, CA 95052

1987/03/09

Distribution List [Pittsburgh/Lesgold] NR 4422539

Dr. Sharon Tkacz
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Douglas Towne
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

Headquarters, U. S. Marine Corps
Code MPI-20
Washington, DC 20380

Dr. William Uttal
NOSC, Hawaii Lab
Box 997
Kailua, HI 96734

Dr. Kurt Van Lehn
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. Beth Warren
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

Dr. Keith T. Wescourt
FMC Corporation
Central Engineering Labs
1185 Coleman Ave., Box 580
Santa Clara, CA 95052

Dr. Douglas Wetzel
Code 12
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Barbara White
Bolt Beranek & Newman, Inc.
10 Moulton Street
Cambridge, MA 02238

Dr. Christopher Wickens
Department of Psychology
University of Illinois
Champaign, IL 61820

Dr. Heather Wild
Naval Air Development
Center
Code 6021
Warminster, PA 18974-5000

Dr. Robert A. Wisher
U.S. Army Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Martin F. Wiskoff
Navy Personnel R & D Center
San Diego, CA 92152-6800

Dr. Dan Wolz
AFHRL/MOE
Brooks AFB, TX 78235

Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Joe Yasatuke
AFHRL/LRT
Lowry AFB, CO 80230

Dr. Joseph L. Young
Memory & Cognitive
Processes
National Science Foundation
Washington, DC 20550

END

DATE

FILMED

MARCH

1988

DTIC