

**NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:**  
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

# COMMUNICATING WITH HIGH-LEVEL PLANS

Technical Report AIP - 49

**Jeffrey Bonar & Blaise Liffick**

Learning Research & Development Center  
University of Pittsburgh  
Pittsburgh, Pa. 15260

May 1988

This research was supported by the Computer Sciences Division, Office of Naval Research and DARPA under Contract Number N00014-86-K-0678. Reproduction in whole or in part is permitted for purposes of the United States Government. Approved for public release; distribution unlimited.

026-2-0214  
No. 49-2-2

**REPORT DOCUMENTATION PAGE**

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution unlimited	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AIP - 49		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Computer Sciences Division Office of Naval Research	
6c. ADDRESS (City, State, and ZIP Code) Department of Psychology Pittsburgh, Pennsylvania 15213		7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, Virginia 22217-5000	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Same as Monitoring Organization	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0678	
8c. ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS p4000ub201/7-4-86	
		PROGRAM ELEMENT NO. N/A	PROJECT NO. N/A
		TASK NO. N/A	WORK UNIT ACCESSION NO. N/A
11. TITLE (Include Security Classification) Communicating with High-Level Plans (Unclassified)			
12. PERSONAL AUTHOR(S) Bonar, Jeffrey and Liffick, Blaise			
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM 86Sept15 to 91Sept14	14. DATE OF REPORT (Year, Month, Day) 1988 May 20	15. PAGE COUNT 20
16. SUPPLEMENTARY NOTATION Workshop on Architectures for Intelligent Interfaces			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	Intelligent interfaces, Plan-based interfaces, Computer interfaces, Plans	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) We discuss our experience with an interface that gives users the ability to directly represent and manipulate goals at several levels of detail. The interface is built into Bridge, a tutorial environment for novice programmers. The name comes from our intended "bridge" between novice and expert conceptions of programming. In order to understand student designs and partial programs, Bridge provides languages that allow a student to talk about his or her high-level designs and partial work. We call the vocabulary of these languages plans. Plans are bundles of knowledge about the standard subtasks in a domain, designed and organized based on a typical user's point of view.			
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS		21 ABSTRACT SECURITY CLASSIFICATION	
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Alan L. Meyrowitz		22b TELEPHONE (Include Area Code) (202) 696-4302	22c. OFFICE SYMBOL N00014



## 1. Introduction

We discuss our experience with an interface that gives users the ability to directly represent and manipulate goals at several levels of detail. The interface is built into Bridge, a tutorial environment for novice programmers [Bonar88]. The name comes from our intended “bridge” between novice and expert conceptions of programming. In order to understand student designs and partial programs, Bridge provides languages that allow a student to talk about his or her high-level designs and partial work. We call the vocabulary of these languages *plans*<sup>1</sup>. Plans are bundles of knowledge about the standard subtasks in a domain, designed and organized based on a typical user's point of view.

Many intelligent interfaces monitor low-level user actions, attempting to infer higher level plans. These inferences are typically implemented with partial matching schemes, based on a plan catalog (see, for example, [Johnson86]). The inferences allow the system to complete user actions, correct errors, or provide tutorial assistance. This approach to inference of user intentions is quite difficult. We propose a different approach, designed to more effectively and accurately capture user intentions.

Our approach gives users a very high-level plan language. By “high-level” we mean a language that is informal, vague, contains much implicit information, and is designed to represent goals of interest to a particular class of users. In particular, the plan language makes assumptions about the user's background knowledge and overall intentions. This is consistent with our interest in providing interfaces to professionals and domain experts who have no programming experience. We focus on users who are experts in a particular task domain and are using a computer to extend or augment that expertise. Our system must take such a user's specification and derive an implementation using the primitives provided by a standard computer system.

Our notion of a high-level programming language – what we are calling a plan language – is quite different than what is used in automatic programming research (see, for example, [Balzer85]). That notion of high-level programming involves highly formal logical specifications. We, on the other hand, are interested in the sort of heuristic, vague, and partially implicit specifications used by humans with other humans.

In the rest of the article we begin by developing a framework for approaching intelligent interfaces. In particular, we discuss the dilemma of a very high-level programming language intended for use by experts who are not programmers. The dilemma arises because interfaces for non-programmers should be both:

- intuitive, that is, understandable based on previous experience, and
- suggest uses that go beyond the limited set of capabilities implicit in an expert's previous experience in a domain.

These two goals seem to be contradictory.

After providing an overall framework, we describe the Bridge programming tutor and a specific implementation of a high-level plan language as used in the Bridge programming tutor. We focus on a

---

<sup>1</sup>The word “plans” is used here in the sense of “a method for achieving an end; an often customary method of doing something” [Websters75].

representation scheme used to describe the plan language in Bridge. We conclude by discussing a new system that provides a high-level plan language interface to a spreadsheet.

## 2. The Dilemma of Intelligent Interfaces

Upon first consideration, one would think to design an intelligent interface to present as intuitive a task as possible to the user. The interface should present the computer system so as to allow the human user to think in exactly the way he or she is used to thinking. Each feature of the system should be presented using terms and conventions familiar from previous experience working without the computer-based support. The advantages of this approach are obvious: the human user begins to use the computer system with little or no training. The features of the computer system are exactly as the user would expect – straightforwardly understandable by appealing to earlier experience.

While having appeal, the “intuitive” approach presented above has an obvious drawback. An interface that merely matches the user’s expectations is stuck with those expectations. In particular, the user can never go beyond those expectations to use more powerful facilities than that expectation allows. Consider, for example, implementing a word processor as an “computerized typewriter.” Slavish attention to this metaphor would give a word processing tool with little of the power we expect from word processors. We would, for example, need to type <RETURN> at the end of every line, use a special brush-like mouse cursor to “white out” mistakes, and need to insert a new diskette at the end of every page. Similarly, if computer based spreadsheets merely provided a convenient grid for laying out numbers – the role of manual spreadsheets – they would not be best selling personal computer applications.

The dilemma, then, is between usability and functionality. Intelligent interfaces should make an application easier to use and understand. Merely applying the intelligence to anticipating and matching every expectation of the user leaves the system doing no more than the user already expects it to do. The question is: how can we build interfaces that allow graceful progression from what is already known to more sophisticated use of a system?

## 3. Our Approach

We approach the intelligent interface dilemma by allowing a large, knowledge rich, and highly specialized set of automated operations called plans. Each set of plans is customized to a particular class of users – managers, secretaries, and statisticians, for example. A total beginner uses these plans as black-boxes, with no understanding of their internal construction. This is possible because the plans have been designed to be simple, intuitive, and directly applicable to particular tasks of interest to that user.

For example, a manager might be given a set of plans that allow him or her to manage projects. For each project, the plans would provide tools for representing subtasks, personnel, applicable resources, deadlines, and etc. The plan set provides fixed capabilities for organizing the project elements, modeling changed deadlines, and creating reports. The details of these capabilities are based on the standard kinds of organizations, models, and reports used by that manager or organization.

Inevitably, the appeal of standard operations wears off. Our user may now be more sophisticated through experience with the plan set. Many users are likely to be frustrated with the lack of flexibility inherent in the fixed set of plans. This is the position of normal experienced users of most

computer software – stuck with the system that was delivered. Even if the system was very well designed, it cannot be tailored to the particular evolving needs of each user.

We propose a new mode of use. Consider that each of the plans provided to the manager is constructed of a few slightly lower level, slightly less powerful plans. The icons that represent the initial set of operations can literally be “opened up,” presenting a small network of the lower-level icons, connected together to describe the behavior of the icon that was opened (see Figure 1). If a user is dissatisfied with a particular form of the high-level icon, he or she can easily open up the icon and redesign it’s behavior with the lower-level icons. Essentially, the user has just created a new operation that is now available for his or her use. In the project management tool example, the manager might modify the way project tasks are described, or how resources are allocated to a project.

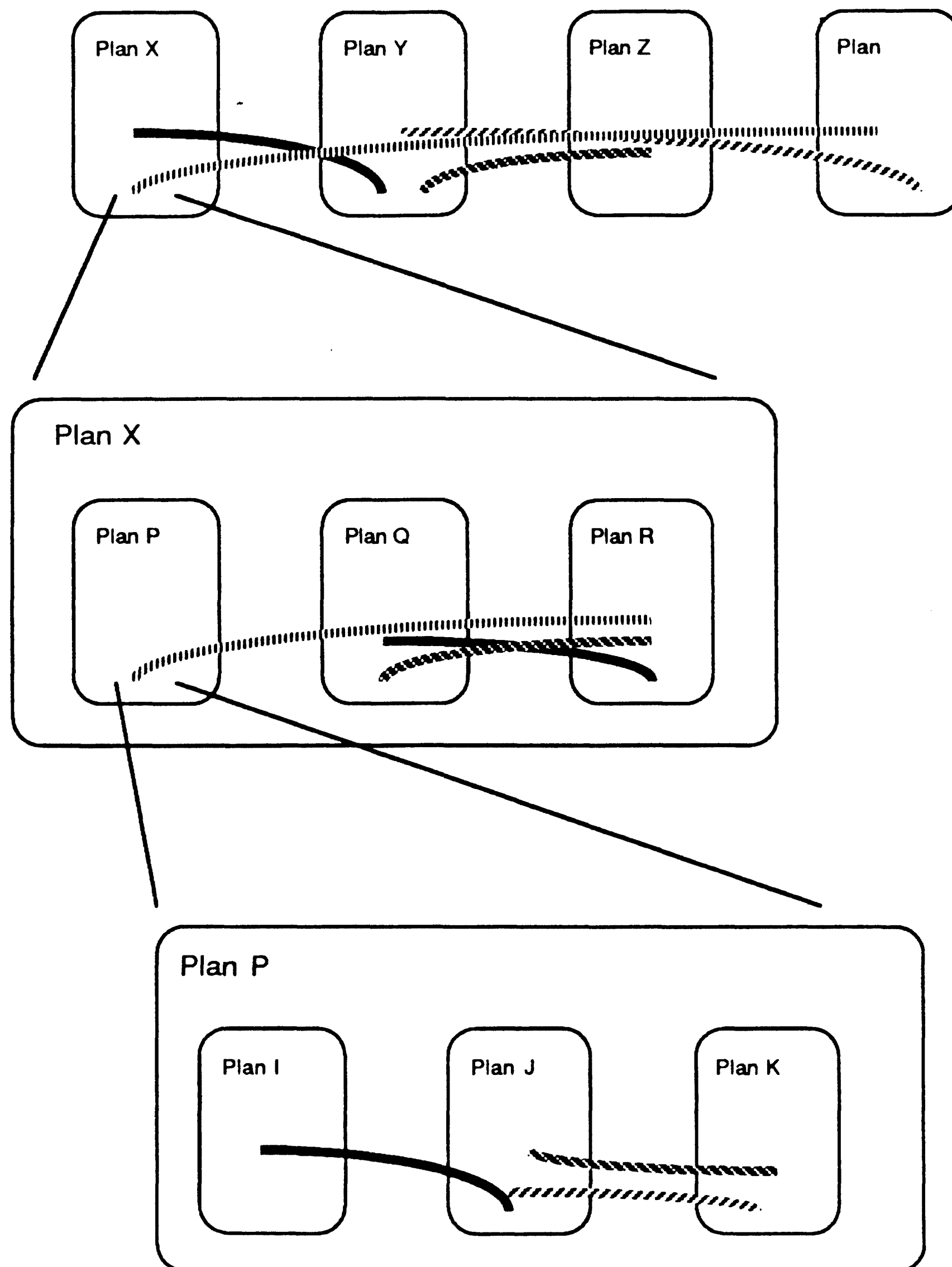


Figure 1. A schematic of the interface design approach advocated in this article.

The user need not do this sort of modification. It is only done when the user is sufficiently dissatisfied with the current plan set to be willing to do the work required to understand the next



level of complexity. If the plan sets are carefully designed, opening up an plan should correspond to exactly one more level of flexibility and complexity. That is, there is exactly one new thing that must be learned in order to master the lower level operations. In practice, there will be many levels of plans defined by lower level plans. By providing many levels, we can give users a smooth continuum from “easy to use” and intuitive to sophisticated and flexible. In essence, the user can become as sophisticated a computer user as he or she desires.

An approach like this nicely meets the needs of novice and casual users. Few users are willing to sit still to learn a sophisticated system. Fewer still are willing to start with the most basic programming elements and build up solutions to real tasks from scratch. Our approach begins with intuitive solutions to real world tasks and allows a user to learn only as much as is required to customize those solutions to the user's needs.

The approach outlined above is sufficiently high-level and vague to evoke little disagreement. Important questions like “how are plan contents determined?” and “how are the various plan links, including the ‘open up’ link, implemented?” are not addressed. While we have no general answer to those questions, the rest of the article presents two specific examples to illustrate how the system might work. One example is drawn from an intelligent tutor to teach programming in Pascal. The second example is drawn from an intelligent spreadsheet tutor currently being implemented in our lab.

## 4. The Bridge Tutor

Research into how novices learn programming reveals that understanding the semantics of standard programming languages is not the main difficulty of novice programmers. Instead, 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, [Eisenstadt81, Mayer79, Soloway84].) In Bridge we built a programming environment that supports a novice in working with plans that describe the goals and subgoals typical of programming tasks. By using plans that describe programming goals, Bridge allows for initial novice conceptions of a problem solution that are informal and sketchy. The Bridge environment features an iconic plan programming language with an editor facilities to control execution and support debugging. A complete discussion of Bridge can be found in [Bonar88].

Bridge supports a novice in the initial informal statement of a problem solution, subsequent refinement of that solution, and final implementation of the solution as programming language code. This is accomplished in three phases, discussed in detail in the rest of this section. To illustrate Bridge use, we discuss a student working on the Ending Value Averaging problem:

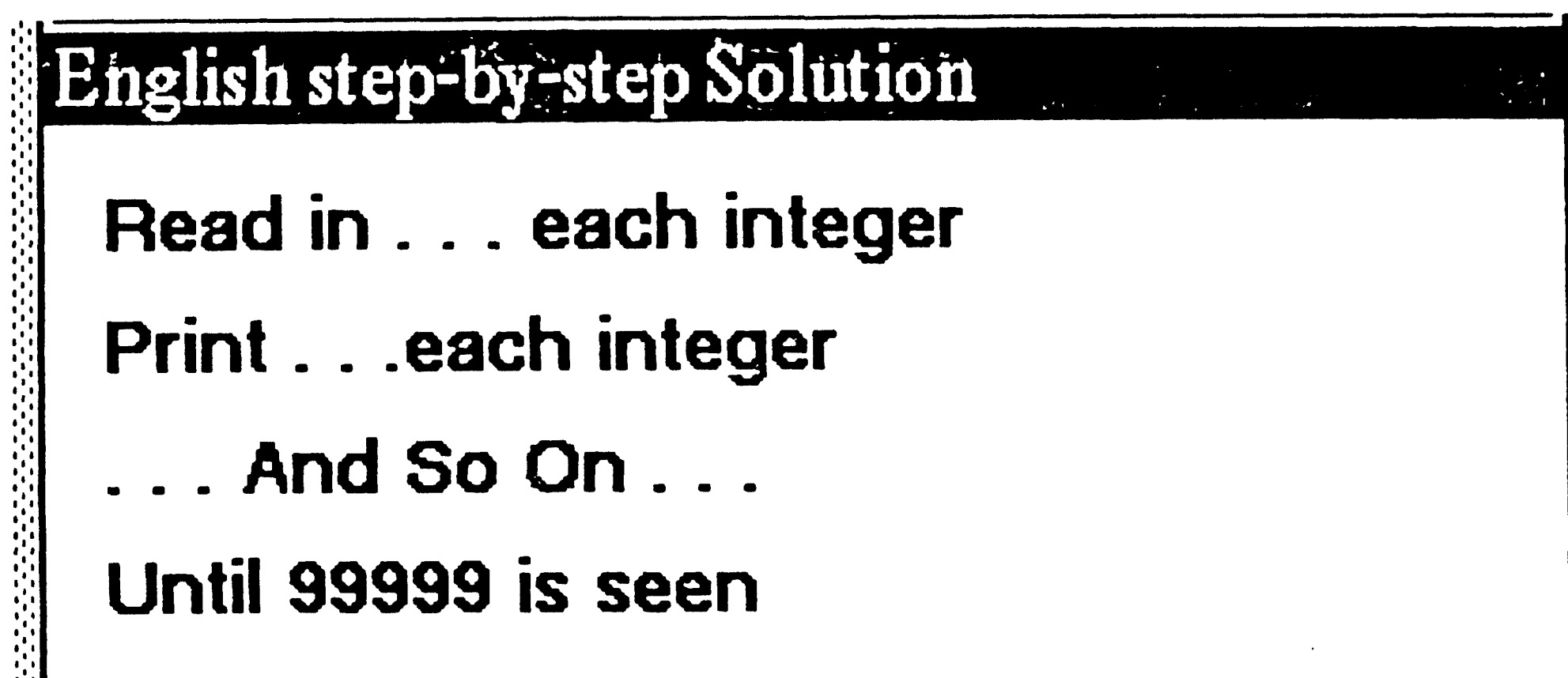
Write a program which repeatedly reads in integers until it reads in the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999.

Each of the phases is summarized, followed by a discussion of how that phase fits into the framework presented in section 3.

## 4.1 Bridge Phase I: Informal Natural Language Plans

The first phase of Bridge involves an informal statement and refinement of the goals for the code. Empirical evidence [Bonar85, Kahney82] 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.

In phase I we provide a plan language based on simple natural language phrases typically used when people write step by step instructions for other people. For example, a student can construct the phrase “... and so on ... until 99999 is seen.” Figure 2 shows an example with several such phrases. Such phrases represent the highest level at which a student can express intentions to the system. Because of the ambiguity in such phrasings, Bridge must understand the student’s intentions based on several possible naive models of programming. For example, a common naive model of looping allows a student to construct a loop with a description of the first iteration followed by the phrase “and so on.” Based on the particular phrasings constructed by the student, Bridge infers a particular naive model.



**English step-by-step Solution**

**Read in . . . each integer**

**Print . . .each integer**

**. . . And So On . . .**

**Until 99999 is seen**

Figure 2. Phrases from a phase I Bridge solution.

Since Bridge’s task is to teach programming, detection of such a naive model results in a tutorial suggestion from Bridge. The architecture underlying Bridge could as easily respond to the user’s naive model directly without attempting to teach the user the “correct” specification. In this more advisory mode, the system would insist on a more fully developed model only when the user’s specification was incomplete or lacking key details.

In terms of the multi-level plan approach discussed in section 3 and illustrated figure 1, the first phase of Bridge is an implementation of the highest level of plans. Each plan stands for the kinds of operations and mental models normally expressed in English language step-by-step specifications written by non-programmers. These operations are vague and include significant implicit knowledge about the objects being operated on. Only a limited set of links are possible between the plans:

ordering and nesting. Data communication links between the plans are implicit, reflecting the structure of natural language specifications.

### 4.2 Bridge Phase II: Iconic Programming Plans

In the second phase of Bridge a programming student refines the informal description of phase I into a series of semi-formal iconic programming plans. Figure 4 shows the Bridge screen as the student is working in phase II. In this phase the plans are schema-like structures which describe how goals are transformed into actual programming code (see [Soloway84] for a detailed discussion of these plans).

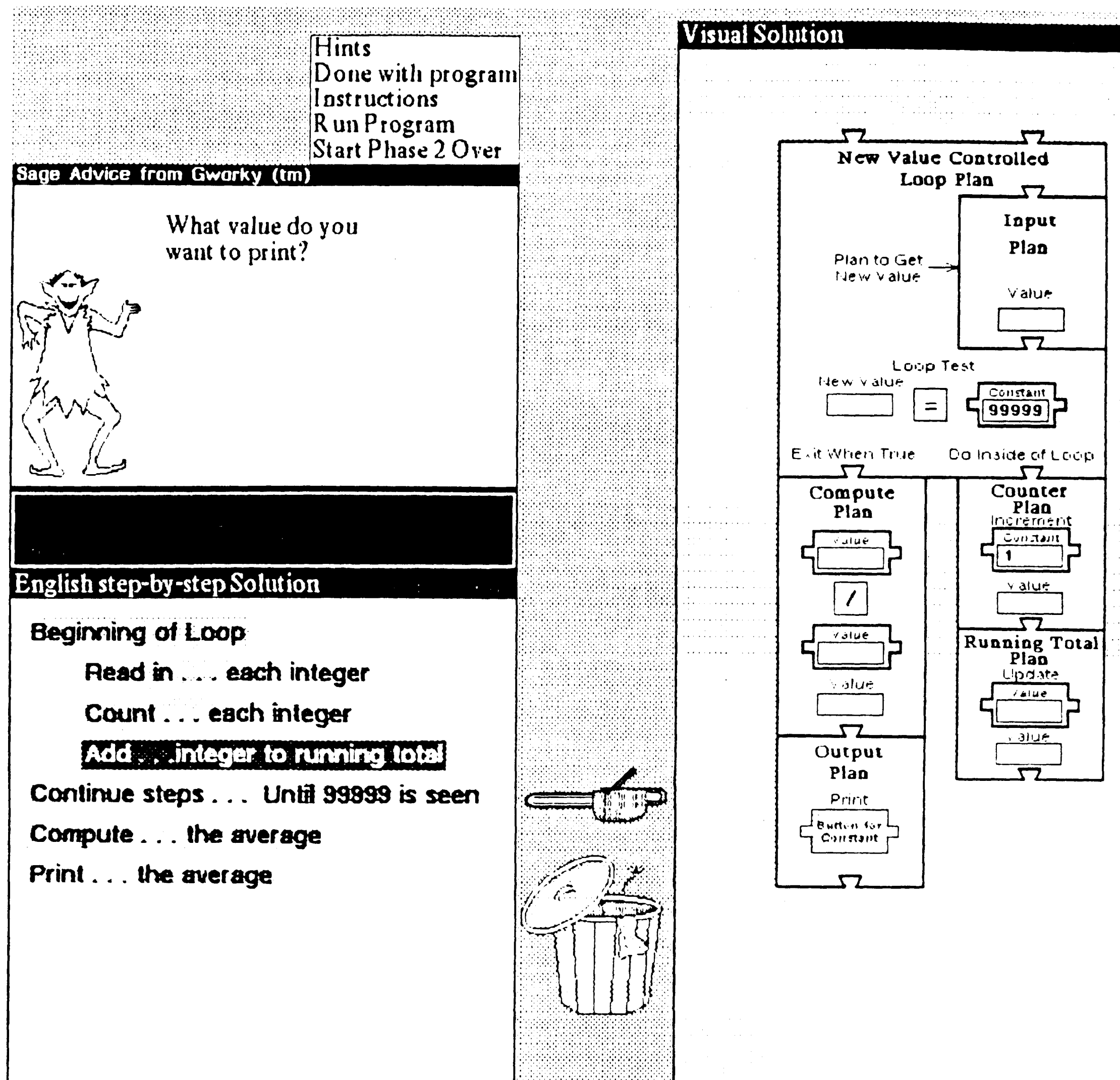


Figure 3. The Bridge system while a student is constructing a phase II solution, based on a complete phase I solution.

Plans have various elements that interrelate with the elements of other plans. So, for example, a counter plan has *initialize*, *increment*, and *use* elements, each with a particular relationship to the loop containing the counter. This interaction of elements often results in the plan implementation in standard programming constructs being dispersed across the program. 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. [Spohrer85] have shown that plan to code translation errors account for many student errors.

In the second phase of Bridge students focus on relating various plan elements, but without compromising the fundamental plan structure and introducing the syntactic complexity required by standard programming code. Figure 4 shows a typical phase II solution to the Ending Value Averaging problem. Each plan is represented by a single icon. There are two kinds of links between the plan elements. Control flow links are expressed by attaching puzzle tabs to puzzle slots. Data flow links are expressed by moving the small tiles with “ears” from the data source to the data destination. This is the way, for example, that the value of the Counter plan is associated with the average computation in the Compute plan.

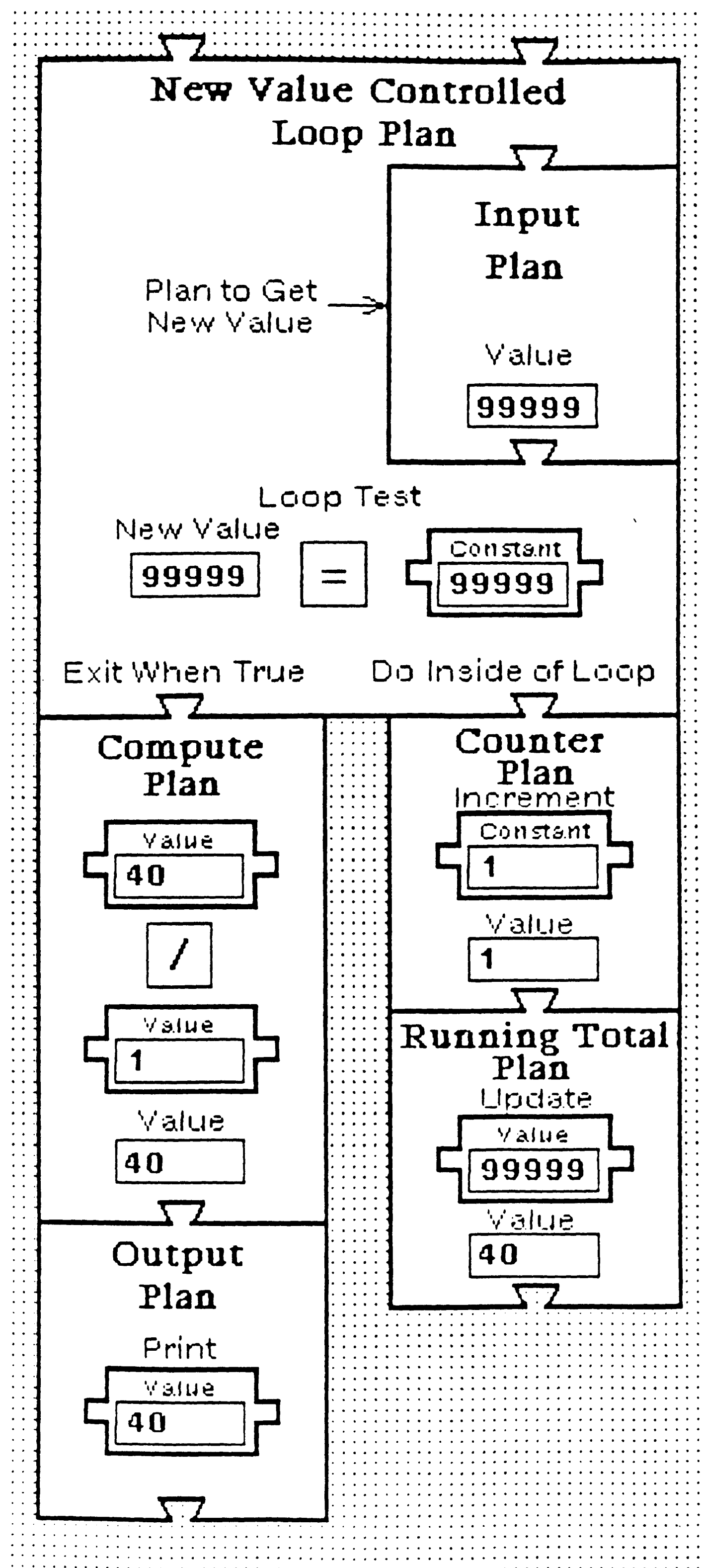


Figure 4. A typical phase II solution in Bridge.

Where phase I of Bridge represents the highest level plans, closest to the understanding of a non-programmer, phase II represents a lower level of plan. The transition from phase I to phase II, then, represents an implementation of “opening up” top-level (phase I) plans. In the current Bridge system the student must complete a phase I plan specification and then must “open up” that specification to implement a phase II elaboration of the original specification. As the student works in phase II, links back to phase I are always available. In a non-tutorial situation, the phase II plan language would allow for refinement of the informal plans specified in phase I, but without requiring the complexity of a specification of the actions actually performed by the machine. The links in phase II express potential connections between elements of the plans, as described above.

### 4.3 Bridge Phase III: Pascal Code

The third phase requires the Bridge student to translate the plan-based description of phase two into actual Pascal code. Students are provided with a Pascal structure editor (much like that of [Garlan84]), and an interpreter with a stepping mode. In this phase the user drops from the world of plans into a standard programming language. In a non-tutorial version of the interface, this phase would be omitted.

### 4.4 Diagnosis with Plans

Any intelligent interface needs to infer user intentions from user behavior. In particular, the system must infer all mental activity from the actual actions performed by the user. In tutoring programming, for example, a standard intelligent tutor must reconstruct the student’s entire mental activity between seeing a program specification and actually entering code in the machine. Such a reconstruction must account for both the correct and incorrect knowledge used by the student during design and implementation.

A tutor’s reconstruction of a student’s program is based on at least two kinds of knowledge. First, there must be a way for students to break the high-level goals of the problem statement into lower level goals. Second, there must be knowledge about how to translate low level goals into program code. Within the tutor we can represent these two kinds of knowledge as operators that transform one kind of structure into another. In addition to correct versions of these operators, the tutor must contain buggy versions representing common student misconceptions.

Using the knowledge base of correct and buggy operators a tutor can, in principle, use search techniques to reconstruct plausible accounts for errorful and ambiguous student specifications. This approach has been powerfully demonstrated in the programming tutor PROUST [Johnson86]. While the approach works, it is very costly in terms of both search time and knowledge engineering. The accomplishments of PROUST must be weighed against the large cost in knowledge engineering time – several hundred hours per problem tutored [Johnson86b]. This knowledge engineering is particularly cost ineffective because the student sees so little of the results. That is, almost all of the knowledge engineering that has gone into capturing operators that describe how to apply programming knowledge is never seen by the student. Inside of PROUST, these operators are optimized for the the search task. They are not available in a form that could be presented to students, or used to assist students as they work towards a solution.

Bridge uses a different approach to reconstructing the student’s intentions. Instead of attempting to reconstruct a student’s entire reasoning from problem statement to final code in one step, Bridge has the student prepare intermediate solutions in plan languages that correspond to particular levels in the

process of moving from problem specification to goals to code. This alleviates many of the difficulties of the PROUST approach. The search is more manageable because it has been broken up into a series of much smaller searches. In each of the smaller searches there are fewer relevant operators to try and less reasoning “distance” to span between the user’s surface behavior and the solution the tutor is trying to reconstruct.

In addition, the Bridge approach simplifies the knowledge engineering. The fewer operators in each search correspond to a smaller overall catalog of operators to be specified. In addition, the smaller search spaces make it easier to tell when the space of possible correct and errorful versions has been covered.

## 5. Plan Language Design

In this section we detail a formal specification for Bridge phase II plans. This specification is not implemented in the current version of Bridge. Instead, it is derived from the experience with Bridge. It is our first attempt to create a plan language that formally captures vague, heuristic, and informal plans. The plan language supplies a systematic semantics that describe how plans execute, how new plans are designed, and how plans are translated into standard programming constructs.

### 5.1 Goals for the Plan Language

As detailed in the previous section, Bridge allows a student programmer to work by describing successively more complex plan combinations. There are four main objectives for the plan languages that express these combinations:

- (1) The plan language should allow users to “connect” to their preconceptions about the domain. Unless users can recognize how their own understanding of a task fits into the planning language, they will find it impossible to formulate usable specifications.
- (2) The plan language should support users in learning a “vocabulary” of plans. Eventually, the user would begin to think in terms of the plans themselves, not the informal plans. Not only is there a catalog of plans, but students are able to create their own plans. This learning will be transparent and effortless, primarily because the plans present a more effective set of distinctions for dealing with a problem.
- (3) When appropriate, the plan language should support novice users in learning how to implement plans with lower level plans or actions closer to the primitives of the system being interfaced. Note that we take as given that an intelligent interface will never be as flexible as a user specifying actions directly.
- (4) For those who are learning the primitives of the system being interfaced, the plan language should support the use of plan-like composition. Plans can be seen as the essence of good design: building blocks that are easily read and understood. We want the users of the system to gain an appreciation for using these plans.

## 5.2 Plan Representation

We begin with the iconic plan language shown in Figure 4. The figure shows a series of plan icons connected together to produce the average of a series of numbers read from the user, stopping with the value 99999. Each icon represents a single plan. Icons are constructed to suggest puzzle pieces, connected together in a finished program. Control flow is suggested by connected pieces, moving in a top to bottom order. Data flow is suggested by moving a “value” icon from the plan supplying a value to the plan using that value. For example, in Figure 4 a value icon is moved from the Input plan to the “update” slot of the Running Total plan. More details on the visual plan language appear in [Bonar88b].

Our 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. Figure 5 shows a piece of the formalism describing the iconic plans in Figure 4.

---

### Running Total Plan

#### *ParentClass*

Loop Action Plan

#### *Slots*

Total:            *value*  
 Addend:  
 Initial:           *initially 0*

#### *Initialization*

Total ← Initial

#### *Execution*

Total ← Total + Addend

### Constant Running Total Plan

#### *ParentClass*

Running Total Plan

#### *Slots*

Increment:        *class Integer, rename Addend*

### Counter Plan

#### *ParentClass*

Constant Running Total Plan

#### *Slots*

Count:            *rename Total*  
 Increment:        *constant 1*

---

Figure 5. A formal description of Running Total and Counter plans.



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. If this section is missing, there is only the default parent class to the plan.

**SLOTS** - each plan can have zero or more slots, which specify data or plan links. One data slot can be distinguished as the “value” slot for this plan. A plan’s value can be used by other plans. So, for example, the value slot of the running total plan is used in the computation of an average. The slots provide a method for referring to other plans. Slots are used, for example, to refer to the plans executed in the body of a loop.

**INITIALIZATION** - this section contains executable code that is performed once when control first flows through the plan. As a special condition, when control flows through a loop plan, it fires its initialization section and the initialization sections of all plans contained in its body.

**EXECUTION** - this section contains executable code that is performed whenever control flows through the plan.

The code given in the **INITIALIZATION** and **EXECUTION** sections is expressed in a simple pseudo-code. This code can easily be translated into a standard programming language given the plan representations and the pseudo-code. The user can have access to this code, allowing them to examine the way a plan is implemented in a standard programming language.

### 5.3 Inheritance

Each plan must specify a parent class explicitly. 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 5 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. Note that the **CONSTANT RUNNING TOTAL** plan has also redefined the inherited **ADDEND** slot (**ADDEND.INCREMENT**) to specify a constant. The **COUNTER** plan further constrains this plan to default to the value 1.

The **INITIALIZATION** and **EXECUTION** sections can also be inherited. In this case, both are inherited by successors of the **RUNNING TOTAL** plan. The renaming and redefining mechanisms specify that the **COUNTER** plan will inherit

$$\text{COUNT} \leftarrow \text{COUNT} + 1$$

without explicitly defining it this way. In addition, it allows us in conversation with the user to describe the “1” as an “increment” rather than as an “addend.”

## A More Complex Example

Figure 6 shows the representation of the ENDING VALUE AVERAGING LOOP (EVAL) plan and its components. This plan is used to calculate the average of a series of values entered by the user.

---

### Ending Value Averaging Loop Plan

#### Slots

Loop	<i>use Sentinel Loop Plan</i>
Body	<i>use Running Total Plan(import Total), use Counter Plan(import Count)</i>
Average	<i>use Average(export Total,Count, import Average)</i>
Output	<i>use Output Plan(export Average)</i>

#### Execution

```
Execute ⇒ Loop
Execute ⇒ Average
Execute ⇒ Output
```

### Sentinel Loop Plan

#### ParentClass

New Value Controlled Loop Plan

#### Slots

NewValue	<i>use Input Plan(import)</i>
Sentinel	<i>class Integer</i>
Test	<i>use Test Plan(export NewValue, Sentinel, NotEquals, import)</i>

### New Value Controlled Loop Plan

#### Slots

NewValue
Test
Body

#### Execution

```
loop
  Execute ⇒ NewValue
  Execute ⇒ Test
  if not Test then exit
  Execute ⇒ Body
endloop
```

---

Figure 6. A formal description of the loop in the Ending Value Averaging problem.

In this example, the LOOP, BODY, AVERAGE, and OUTPUT slots contain links to other plans, using a procedure header style mechanism for referring to other plans. Note that it is possible to pass

values to linked plans. For instance, in the AVERAGE slot, the AVERAGE plan is passed the values of TOTAL and COUNT with which the AVERAGE plan will compute a value for the slot. The OUTPUT slot sends this average to the OUTPUT plan.

The EXECUTION section of the EVAL 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.

Note that the LOOP slot contains a reference to the SENTINEL LOOP plan. This SENTINEL LOOP plan is a child class of the NEW VALUE CONTROLLED LOOP plan. The SENTINEL LOOP plan does not have any executable sections, but does define three new slots that are used during execution.

Finally, note that the code for performing the actual looping action appears only in the NEW VALUE CONTROLLED LOOP plan. The EVAL plan and the SENTINEL LOOP plan leave this detail to another level so that it can be 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 child class 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 EVAL plan that should define that the body of the loop contains a RUNNING TOTAL and a COUNTER. The SENTINEL LOOP plan is not responsible for this detail, since its only concern should be how to construct a loop that repeats until some sentinel value is reached.

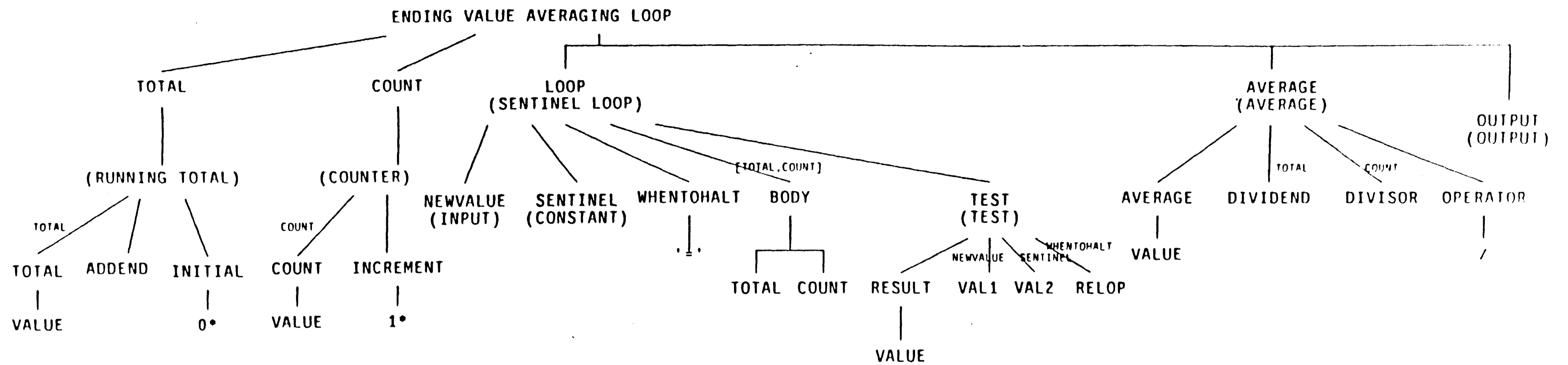
## 5.5 Using the Plans

Initially we intend that a novice would use plans at the highest possible level, constructing programs by selecting from a menu of plan icons, and connecting the icons together to produce a problem solution. However, it is intended that students examine the plans themselves, either through curiosity or need. Eventually, the student will wish to modify existing plans, creating customized versions for his/her own needs.

This is all consistent with the goals expressed above. In addition, such manipulation of the environment is supported by the representation of plans already outlined. The user can navigate the hierarchy of plans in two distinctly different ways. Figure 7 shows a graphical representation of the ENDING VALUE AVERAGING LOOP plan and its components, as derived from the formal plan descriptions. The student is able to navigate such a structure and, as his/her ability grows, modify the contents of the plans identified.

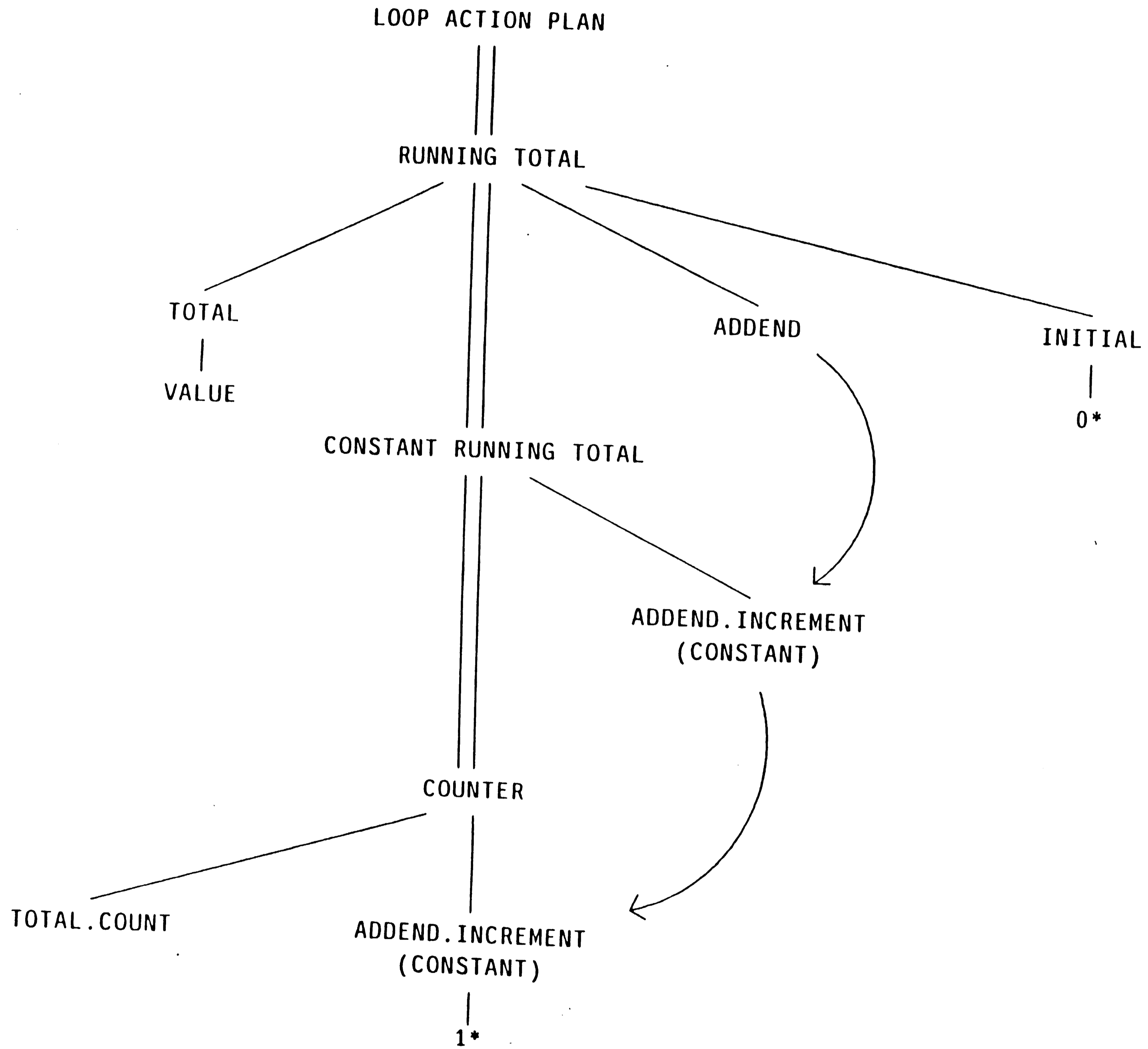
A second way that the plans might be navigated, changing to a different point of view, is through the class hierarchy. Figure 8 shows how this might also be represented graphically. By navigating this representation, the user moves from more specific levels (such as the COUNTER plan) to more general ones (up to, for instance, the RUNNING TOTAL plan). He/she can also examine how slots have been renamed or redefined (as in the case of the ADDEND.INCREMENT slot).

The hierarchy of Figure 7 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 plans. Traversing these levels also lets users explore the plan taxonomy. In this way the user gains a more sophisticated view of the programming environment and available programming tools.



\*DEFAULT

Figure 7



\*DEFAULT

Figure 2

## 6. A Spreadsheet Plan Language

We are generalizing the Bridge Plan system to address issues in the development of a spreadsheet tutor. With the spreadsheet system we are attempting to build a plan based intelligent interface whose focus is not on teaching, but on intelligent support for a expert who is not experienced with spreadsheets.

Traditionally, a spreadsheet is an oversized piece of paper made up of rows and columns intersecting to form boxes which contain words or numbers. Functionally, the spreadsheet serves as a structure for performing financial calculations, testing assumptions, and analyzing the results. Electronic spreadsheets also consist of intersecting rows and columns, but stored and displayed on a computer. Certain cells are defined with a formula that references other cells. The cell displays the value computed by the formula. Any change in a cell referenced in the formula causes the formula to be recomputed and the new value displayed. Computer based spreadsheets allow for effortless changes. When one number on the spreadsheet is changed, either to make a correction or perform some analytic forecasting function, many other values might depend upon it and therefore also need to be changed. What would be a very tedious job by hand can be done with speed, accuracy, and ease on a computer.

The typical user of a spreadsheet is an expert in some domain, typically a business or engineering specialty, and a novice computer user. This strikes us as a very common class of users and a crucial application for intelligent interfaces. The “dilemma of intelligent interfaces” is particularly relevant in this situation:

- To a certain extent, an intelligent spreadsheet wouldn't look like a spreadsheet at all, since the expert has only particular situations and tasks to be accomplished with the spreadsheet. Instead, the system would present a set of basic calculations and capabilities corresponding to these situations and tasks. These could be selected and pieced together.
- On the other hand, we do not want to limit our experts and would like to give them ways to understand and use the full functionality inherent in a spreadsheet.

Spreadsheets are an interesting domain for reasons other than their typical user. A recent study gave six experienced spreadsheet designers four spreadsheet tasks of moderate complexity (2-6 hours). Forty-three percent of the resulting spreadsheets had serious errors in the results predicted by the spreadsheet. This is a troubling result, suggesting that standard spreadsheet languages are too low-level and error prone for reliable use with tasks of even moderate complexity. The plan-based approach to intelligent interfaces discussed in this article can address this problem. For most spreadsheet tasks being done by typical spreadsheet users, it would not be necessary to build low level formula and links. Instead, a series of spreadsheet plans would be assembled. As the plans are assembled, they would compile themselves into standard spreadsheet formulas and load themselves into an actual spreadsheet. Where someone cared to fiddle with details, or was forced by an unusual problem to specify a particular detail, they could interact with the spreadsheet directly, but only in a very local and particular context.

For such a scheme to work, it is necessary that we actually find spreadsheet plans of sufficient generality to apply to the task. To do this, we examined several “how to” book on using spreadsheets in business (for example, [Anderson87, Clark86]). We studied many spreadsheets used for business tasks like income statements, checkbook accounting, sales summary, purchase summary, operating expenses summary, cash flow summary. The study yielded surprisingly few plans in use. The most common plans are summarized in Figure 9. That figure shows a set of plans, grouped at the highest level as Input Table plans, Output Table plans, or Output Report plans.

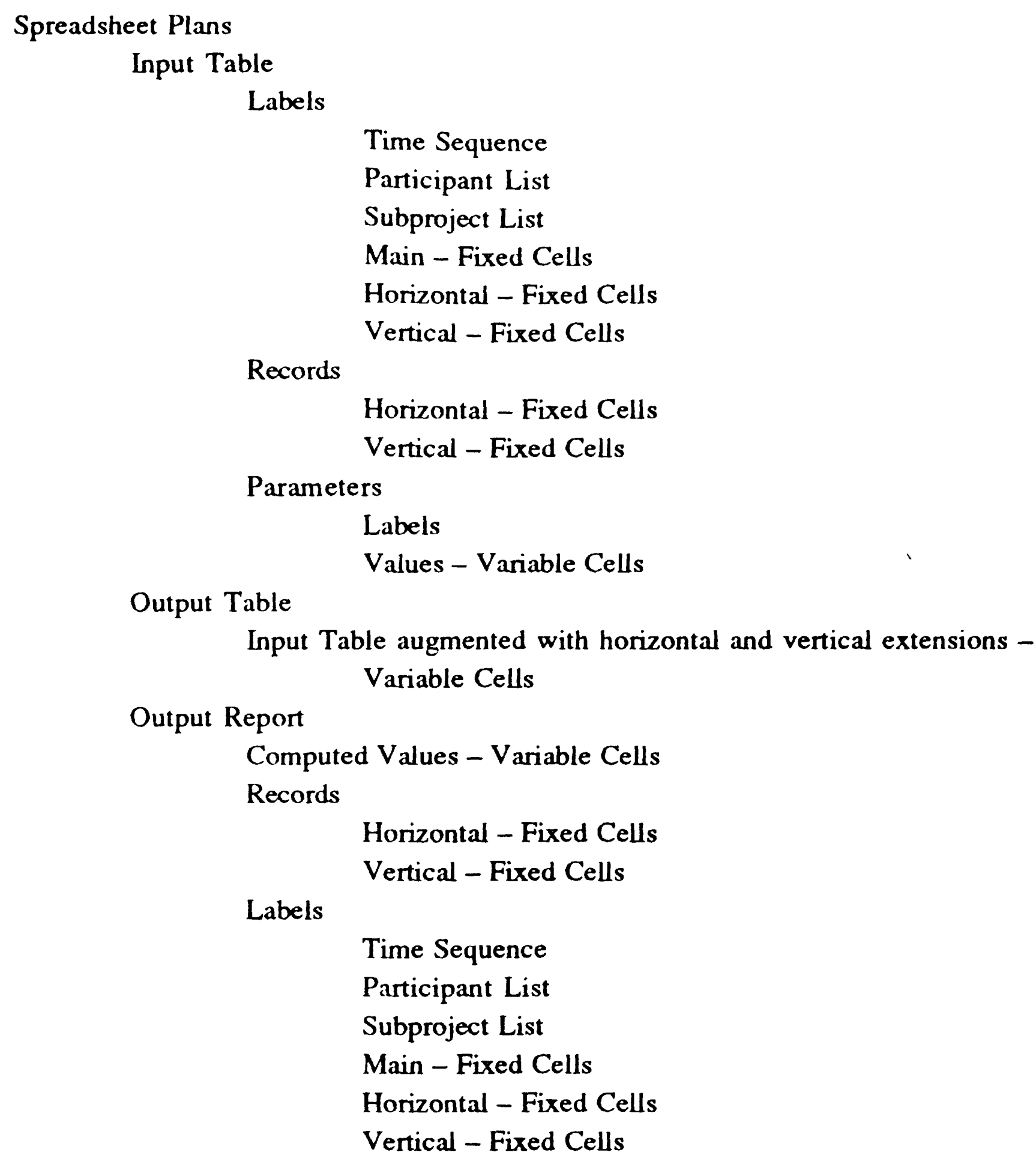


Figure 9. Outline of spreadsheet plans and plan components in use in common business spreadsheets.

The chart in figure 9 shows the plans and plan components found in common business world spreadsheet plans. For example, almost all spreadsheets had an Input Table plan, with axis based on a time sequence, sequence of names, or sequence of subprojects.

Currently, the spreadsheet interface has a crude interface to these plans and allows the user to design a spreadsheet by selecting components off a menu.

## 7. Related Work

The most notable work in the development of a plan formalism is the plan calculus used in the Programmer's Apprentice [Rich81, Shrobe79, Waters78]. The main emphasis of this formalism is the "analysis, synthesis, and verification of programs" [Rich81]. While these are important issues, they are not the concern of our own system. There is little to suggest that the plan calculus can be used

effectively by novice programmers to construct their own plans, to modify available plans, or to understand programming plans themselves.

## 8. Conclusion

We have sketched a formalism that allows novice programmers to program with plans. The formalism neatly specifies the behavior of a high level visual plan language. In addition, the formalism allows students to connect plans to their informal experience, implement plans in a conventional programming language, and define new plans based on previously defined plans. The formalism provides a scheme for realizing a high-level planning language for use as an intelligent interface.

## Acknowledgements

Many thanks to Robert Cunningham for his stunning implementation of Bridge. Our thanks to A. Namasivayam who is implementing the spreadsheet tutor and to Mary Lewis who developed the spreadsheet plan set.

## 9. References

- [Anderson87] Anderson, L., Cobb, D. 1987 *1-2-3 For Business*. Que Corporation, Indianapolis, Indiana.
- [Balzer85] Balzer, R. 1985. A 15 Year Perspective on Automatic Programming. *IEEE Transactions of Software Engineering*, SE-11 (November).
- [Bonar88] Bonar, J., Cunningham, R., Beatty, P., & Weil, W. 1988. Bridge: Intelligent Tutoring With Intermediate Representations. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260
- [Bonar88b] Bonar, J., Liffick, B. 1988. A Novice Visual Programming Language. To appear in *Visual Languages and Programming*, edited by S.K. Chang, Prentice-Hall (in press).
- [Bonar86] Bonar, J., Weil, W., & Jones, R. [1986]. The Programming Plans Workbook. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260
- [Bonar85] Bonar, J., and Soloway, E. 1985. Pre-programming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1.
- [Clark86] Clark, R., Swarsey, P. 1986. *The Compleat IBM Spreadsheets*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.



- [Eisenstadt81] Eisenstadt, M., Laubsch, J., and Kahney, H. 1981. Creating Pleasant Programming Environments for Cognitive Science Students. Proceedings of the Third Annual Cognitive Science Conference. Berkeley, CA.
- [Garlan84] Garlan, D.B., Miller, P.L. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. In Proceedings of the Software Engineering Symposium on Practical Software Development Environments, Association for Computing Machinery.
- [Johnson86] Johnson, L. 1986. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufman, Palo Alto, CA.
- [Johnson86b] Personal communication.
- [Joni85] Joni, S., Soloway, E. 1985. ??
- [Kahney82] Kahney, H., & Eisenstadt, M. 1982. Programmers' mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society.
- [Liffick87] Liffick, B. 1987. A Visual Programming Environment for Novices. Technical Report, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260
- [Mayer79] Mayer, R.E. 1979. A Psychology of Learning BASIC. *Communications of the Association for Computing Machinery* 22(11).
- [Rich81] 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.
- [Shneiderman79] Shneiderman, B., Mayer, R. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences*, 8(3).
- [Shrobe79] Shrobe, H. 1979. Dependency Directed Reasoning for Complex Program Understanding. PhD Thesis. MIT/AI/TR-503, April 1979
- [Soloway84] Soloway, E., & Ehrlich, K. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions of Software Engineering*, SE-10
- [Spohrer85] Spohrer, J., Soloway, E., & Pope E. 1985. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction*, 1.
- [Waters85] Waters, R.C. 1985. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions of Software Engineering*, SE-11
- [Waters78] Waters, R.C. 1978. Automatic Analysis of the Logical Structure of Programs. PhD Thesis. MIT/AI/TR-492, December 1978.

[Websters75] 1975. *Webster's New Collegiate Dictionary*. G&C Merriam Company, Springfield, Massachusetts.

