Journal of Visual Languages and Computing (1992) 3, 299–317

VPCL: A Visual Language for Teaching and Learning Programming. (A Picture is Worth a Thousand Words)

ALIREZA EBRAHIMI

Department of Computer Science, State University of New York/College at Old Westbury, Old Westbury, New York 11568, U.S.A.

Received 22 September 1990 and accepted 1 May 1992

There is a need to incorporate visualization in programming. This visualization can be accomplished through various programming steps such as plan composition, language constructs and program execution. Several empirical studies of programmers reveal that major programming errors are related to plan composition and language constructs. These programming steps are considered in the development of a new visual environment known as VPCL. To understand and learn programming, VPCL is divided into three phases: plan observation, plan integration and plan creation. During the plan observation or elementry level, the programming steps of a plan are rehearsed. In the intermediate level, the plans of a given problem are integrated by the user. In the advanced level, all the programming steps are developed using VPCL tools and the language constructs library. Each phase of VPCL is illustrated in detail with several examples. The effectiveness of VPCL as an instructional and developmental tool is demonstrated by the analysis of a sample empirical study.

1. Introduction

THE VISUAL PROGRAMMING PROCESS can be integrated into the development of a new environment known as Visual Plan Construct Language or VPCL. Visualization is incorporated into the programming process using plan composition, language constructs and program execution (control flow). A plan identifies a specific task, such as a programming problem, which itself may consist of other plans. A plan can be represented by an icon, which can hide the plan's details. Language constructs, such as loops or decisions that are used to implement plans, can be represented visually. The animation of program execution can be used to illustrate the flow of data.

VPCL emphasizes the plan and its manipulation. The user can learn about plans, how to decompose problems into plans, and ultimately how to put plans together to create working programs. It allows new plans to be created or existing ones to be modified. VPCL helps the user to understand different language constructs and their alternatives by providing a language construct library; it is not language-dependent and can thus be adjusted to use the current conventional languages. Starting with a series of programming problems and other utilities known as the plan library, VPCL helps in understanding and developing programs.

VPCL consists of three selectable phases. These phases are rehearsal, partial implementation and full implementation of plans; these may be designated, respectively, as: (1) plan observation; (2) plan integration; and (3) plan creation. The details of these phases will be explained further.

1045-926X/92/030299 + 19 $08.00/0

2. Programming Errors

Empirical studies of programming reveal that programming is not just a matter of the syntax and the semantics of language constructs, rather it is "putting the pieces together" to compose a program. Programmers may understand language constructs such as the conditional **if** or iterative **while** statements, but when it comes to putting them together, they encounter problems.

2.1. Plan Composition Errors

Expert programmers have built large libraries of paradigmatic solutions, known as plans, which can be used to solve new problems [1–6]. A programming problem is a plan which can be decomposed into other plans. In order to give a better understanding of a plan, one can describe it analogically as a kind of theater or play, which itself contains other 'plans' for actors, scenes, etc. This plan representation may include the stage direction for each actor and/or the script for each scene, as well as their various interrelationships.

In programming, a plan may consist of related pieces of code representing a specific action. It can be as short as a single statement or as long as an entire group of codes, which may in itself contain other plans. For example, a plan containing a single line such as C = C + 1 is known as an incrementer plan. A series of scattered lines of code used for declaration, initialization and incrementation to accumulate the number of objects is identified as a counter plan. The entire group of codes forming a programming function (module) called the sort plan is used to sort data. This plan also may itself contain other plans.

Plans should not, however, be mistaken for procedures or functions in programming. A function might be a plan, but a plan is not always a function, since the codes of the plan may be scattered throughout the program. Moreover, plans are independent of programming languages and can thus be coded in different programming languages such as C, Pascal and others.

Figures 1 and 2 demonstrate the plan representation and their integration for an elementary problem known as the 'averaging program'. This program computes the average of a series of input data terminated by a sentinel value 9999.

2.2. Language Construct Misconceptions

Language construct misconceptions have also been one of the major causes of programming errors [4, 7–9]. Most of these errors can be categorized into two types.

1. The programmer's improper translation of natural language expressions into computer programming language, such as the substitution of **and** for **or**, and **while** for **if** or vice versa. For example:

```
if (temp < 0 && temp > 100)
    printf ("system shut down");
```

   The above example is not correct since the temperature cannot be less than 0 and greater than 100 at the same time. In the following 'count characters'

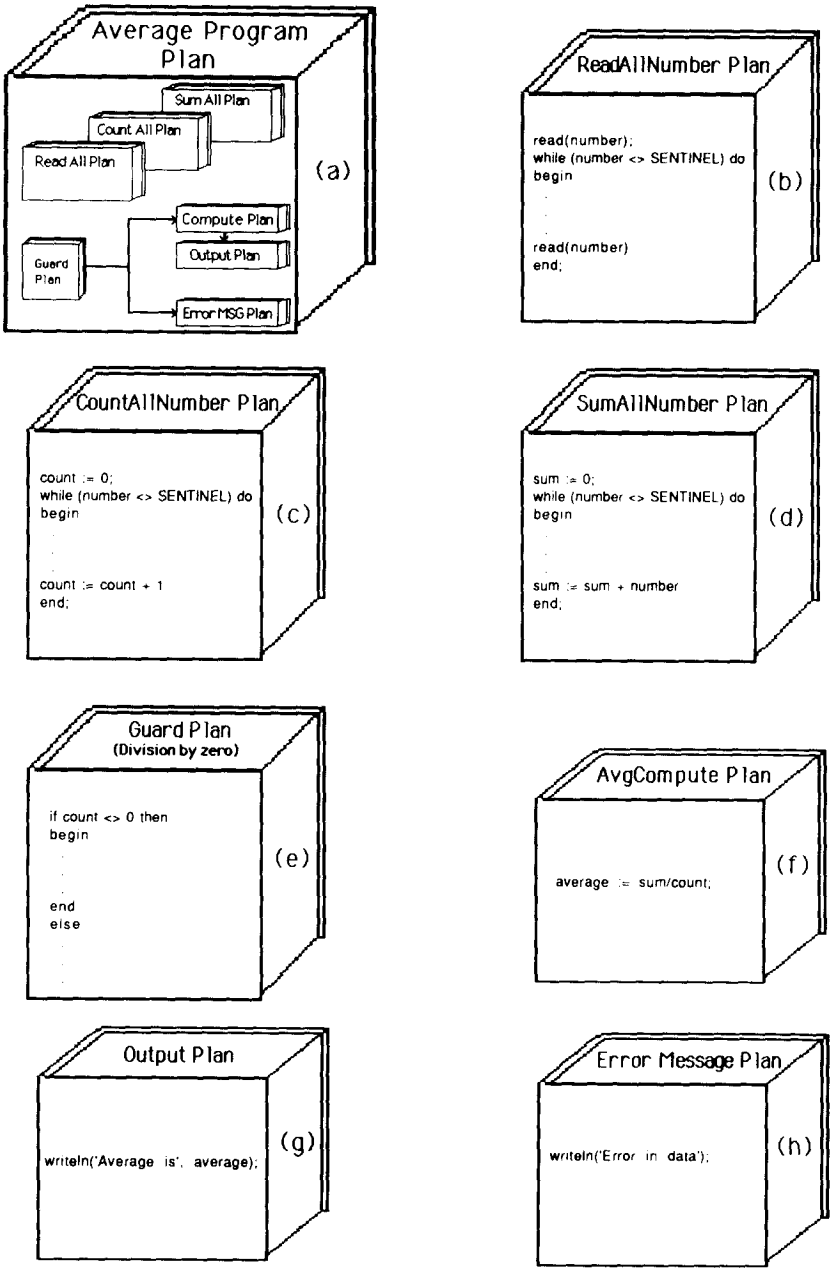

Figure 1. Average problem plan representation (using Pascal). (a) Visual illustration of the plans and their relationship. (b)–(h) Plan representation and programming code

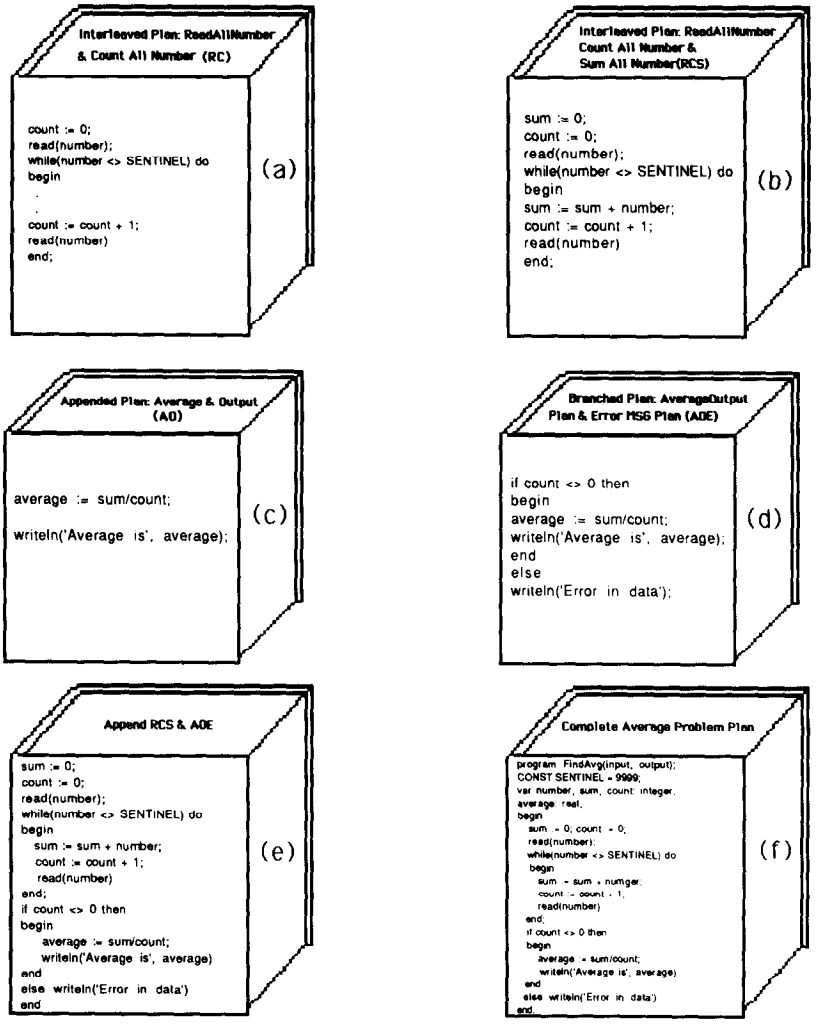

Figure 2. Average problem plan integration. (a)–(e) Plan integration using VPCL mechanism (further discussed in Section 4.2). (f) Complete average program plan

example, the second **while** statement should simply be an **if** statement.

```
while ((c = getchar())! = EOF)

while (c! = NEWLINE) nc++;
```

2. The programmer's misconception caused by arcane or arbitrary notations used in language design such as selection of operators, keywords, data types, etc. For example:
   (i) In the C language the operator '==' is used for the equality test while the operator '=' is used for assignment. An expression like the assignment statement can be embedded in other expressions such as conditional or iterative statements.

When tested, any value in C, except zero, is considered to be true. The following example demonstrates how the user confuses the assignment operator '=' for an equality operator '= ='. Here, the value of the variable 'B' is assigned to 'A' and the value of A is tested.

```
if (A = B) printf ("Both are equal");
    else printf ("not equal");
```

The value of A will be false if B is zero; otherwise A will be true.
Due to the way integer division is handled in C, FORTRAN and other languages, the following Fahrenheit to Celsius formula, $C = 5/9 \times (F - 32)$, will be equal to zero ($5/9 = 0$). Similar problems will arise within the languages that allow implicit data conversion.

The eliminate some of the language construct misconceptions, VPCL provides assistance through a library of constructs with various examples related to common problems. This will be discussed further in Section 5.

3. Visualization and Programming

Different empirical studies of programming suggest that understanding and productivity of programming can be improved by plan composition and/or visualization [4, 5, 10–13]. The improvement and simplification of the teaching and learning of programming were major considerations in the specification and design of Pascal, yet Pascal and other conventional programming languages such as FORTRAN, C and Lisp were limited to linear and textual methods. These languages lacked visualization and an environment to support plan compositions. Today, most computer applications and systems, such as databases and operating systems, have incorporated visualization into their design perspective. The use of images, icons, buttons, windows, folders, fields, colors, sounds and the mouse leads to improvement of productivity. This process can be laid out in the following steps.

1. *Plan representation.* Images or icons can be used to illustrate the plan as an abstraction.
2. *Plan composition.* Integration of plans can be visually demonstrated.
3. *Program execution.* Program execution with data, and control flow in execution can be shown by animation.
4. *Language construct representation.* Constructs such as condition (**if**, **switch**) or loops (**for**, **while**, **do...while**) can be pictorially represented.

4. VPCL: Visual Plan Construct Language

VPCL is based on visualization and thus remedies the aforementioned two weaknesses of novice programmers: plan composition and/or language constructs. VPCL explicitly emphasizes plans and their manipulation, hence reinforcing programming abstraction.

From the very beginning VPCL contains a set of plans known as the plan library. This plan library is exhibited by a bookshelf where each book represents a plan problem that can be selected using a mouse (Figure 3). Original plans are predefined

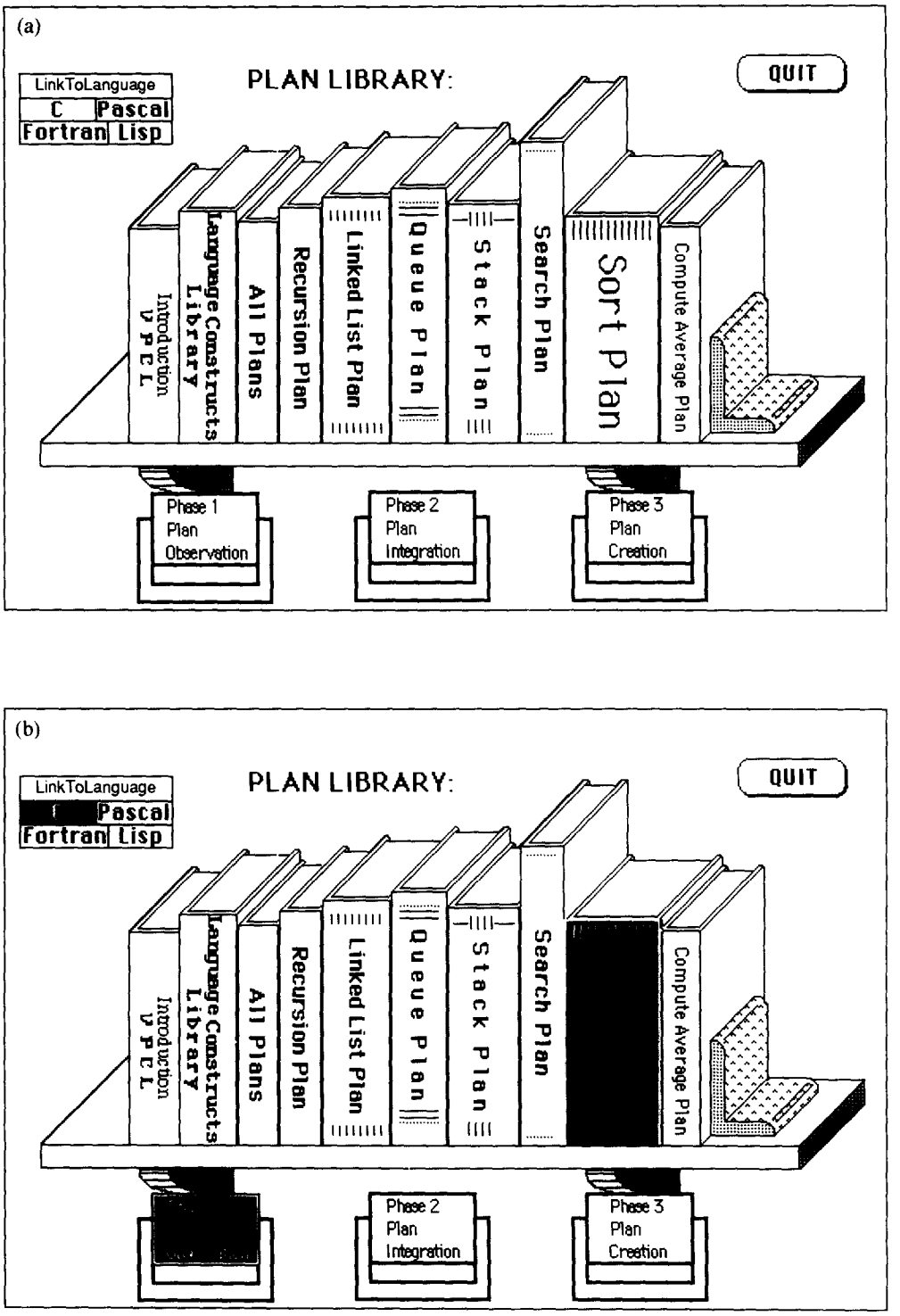

Figure 3. (a) Plan library. (b) Selection of sort plan, phase 1 and C language

and preprogrammed but additional plans can be created and added to the library by the user (Section 4.3).

VPCL is also different from other visual programming languages since it uses visualization in the entire process of programming. The visualization process consists of plans, their composition, program creation and animation of the program execution, along with data. VPCL is divided into the following three phases.

- Phase 1: plan observation (rehearsal).
- Phase 2: plan integration (partial implementation).
- Phase 3: plan creation (full implementation).

The original screen display of VPCL presents a plan library with three phases; each plan can be used in conjunction with any of these phases. Selection of a plan with a phase is illustrated in Figure 3(b). A new plan can be created by modifying an existing plan or by writing an entire plan code.

VPCL with its language construct library assists the user in writing a program. This library provides the syntax of language constructs (frequently visualized) that can be chosen to build a new plan. Within the library there are tutorial sessions, 'learning by example', for each of the language constructs. This will free the user from having to memorize the jargon of the syntax and semantics of a particular programming language construct. As a result, the user will have a better picture of language constructs and their alternatives.

As stated earlier, VPCL is language-independent and it can be adjusted to use the current conventional languages. A plan code can be presented in several languages such as Pascal, FORTRAN, C or even Lisp. VPCL visually animates the process of program execution with data such as control flow of the program and updated variables. Looking towards the future, one can see how VPCL can take advantage of parallel machines where more than one plan is being executed concurrently. VPCL has been implemented using Hypertalk scripting, an object-oriented language with Hypercard utilities on the Apple Macintosh.

4.1. Phase 1: Plan Observation

This phase of VPCL is an automatic animated process, illustrating the steps involved in programming from the initial specification of the problem to the final execution of the program with data. VPCL starts with a screen displaying the plan library shown as a bookshelf where every book is assigned to a plan for a programming task. In order to activate a plan in this phase, both the desired plan and the phase 1 button must be clicked one after the other using a mouse. This is illustrated in Figure 3(a) and (b). After the selection has been made, VPCL starts its journey. It demonstrates the plan composition of the problem where plan relationships can be seen. Embedded plans are further decomposed. When there is no further plan decomposition, the code for each of the plans is generated and displayed in the chosen programming language. Later, these plans are integrated one by one to form the complete program. After the program has been generated, animation of the program's execution is shown.

The following example illustrates phase 1 of VPCL choosing the selection sort of the sort plan from the plan library. This sort orders a series of numbers by finding the smallest number and placing it at the beginning (the sorted place). The same process is repeated with the rest of the numbers until all numbers are sorted.

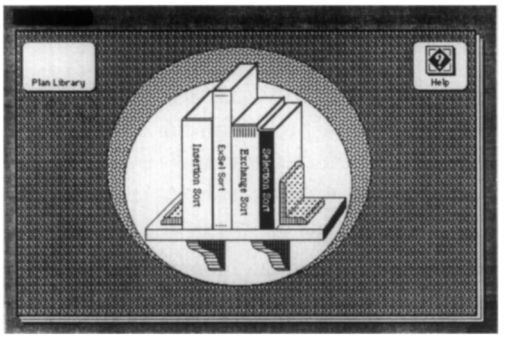

Figure 4. Plan activation, *Selection Sort* chosen

4.1.1. Plan Activation

Sort plan and phase 1 are highlighted from the plan library by clicking the mouse. A window is displayed with the following sorts currently available: *Selection*, *Exchange*, *ExSel*(combined exchange selection) and *Insertion*. In this case the *Selection Sort* was chosen (Figure 4).

4.1.2. Plan Decomposition

The *Selection Sort* is decomposed into the following plans [Figure 5(a)].

1. Input plan, which gets the data.
2. Loop select smallest and exchange plan (LSSE), which finds the location of the smallest value. Later, the smallest value is exchanged and placed in its proper location.
3. Output plan, which prints the sorted output.

Since LSSE consists of embedded plans, it is further decomposed into the following Plans [Figure 5(b)].

- Loop plan.
- Select smallest plan.
- Exchange plan.

4.1.3. Coding of Plans

When a plan cannot be further decomposed, its programming code is displayed in one of the selected programming languages originally chosen from the plan library, e.g. the C language [Figure 5(c)–(f)].

4.1.4. Composition of Plan's Code

To form a complete program, the code of one plan should be combined with the code of another plan. For example, plans for *select smallest* and *exchange* should be appended forming a new plan known as 'SSE' [Figure 5(g)]. This plan is then embedded within the loop plan to form 'LSSE'. Plan code composition will continue until the final program is produced. Mechanisms for VPCL plan composition and detailed explanations are illustrated in phase 2.

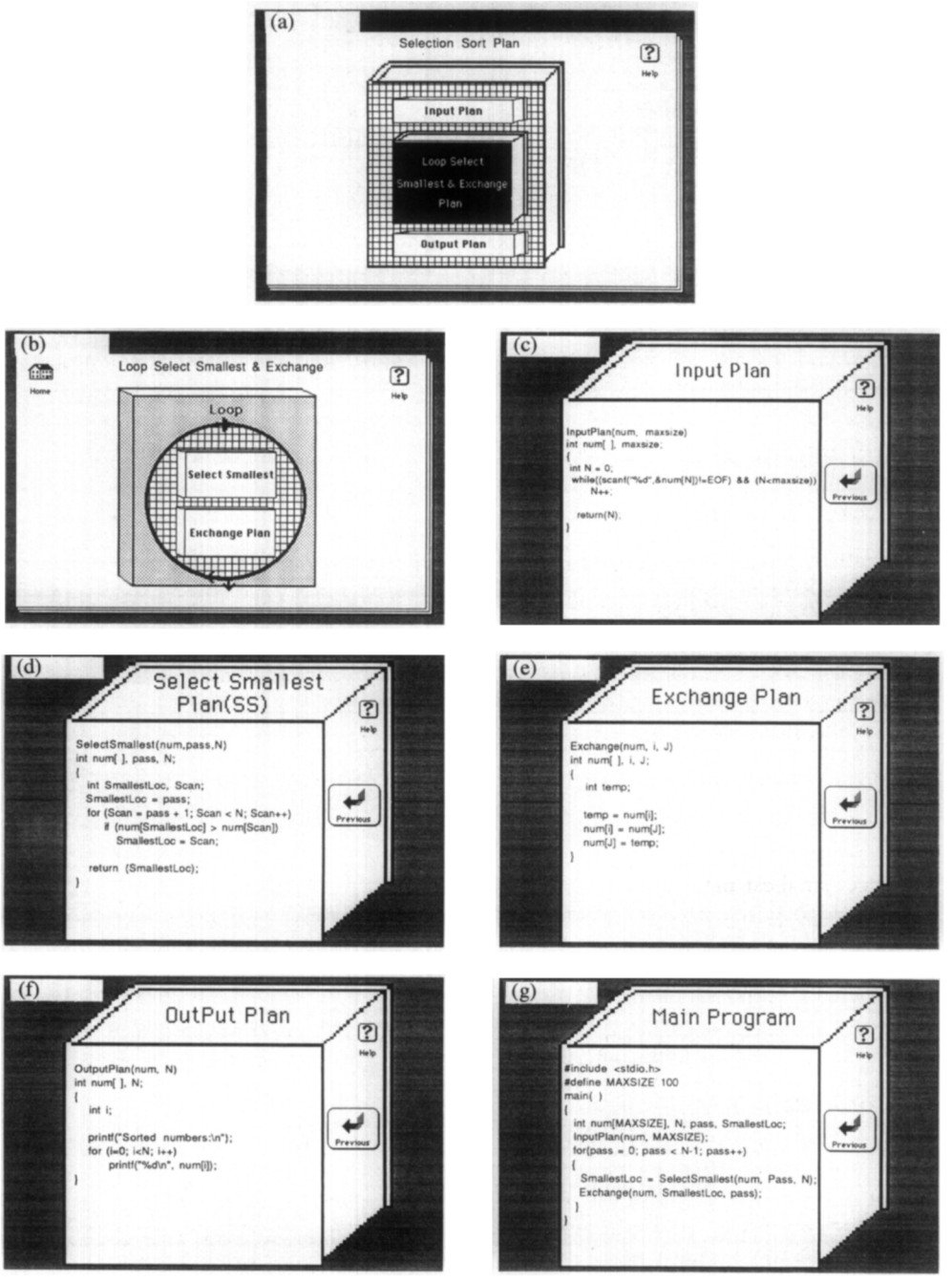

Figure 5. (a) and (b) Plan decomposition. (c)–(g) Plan code

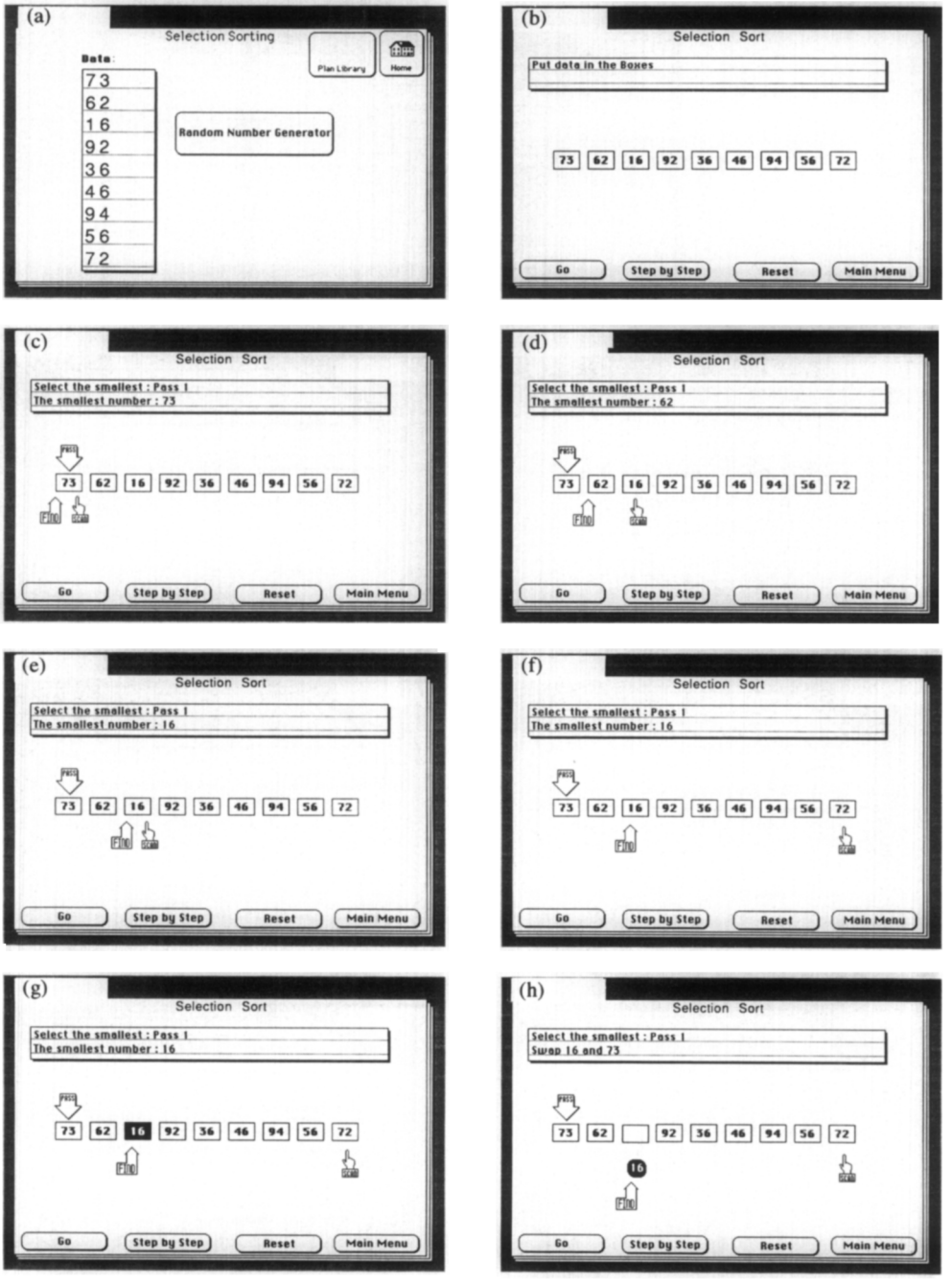

Figure 6. (a)–(p) Animation of program execution for *Selection Sort*

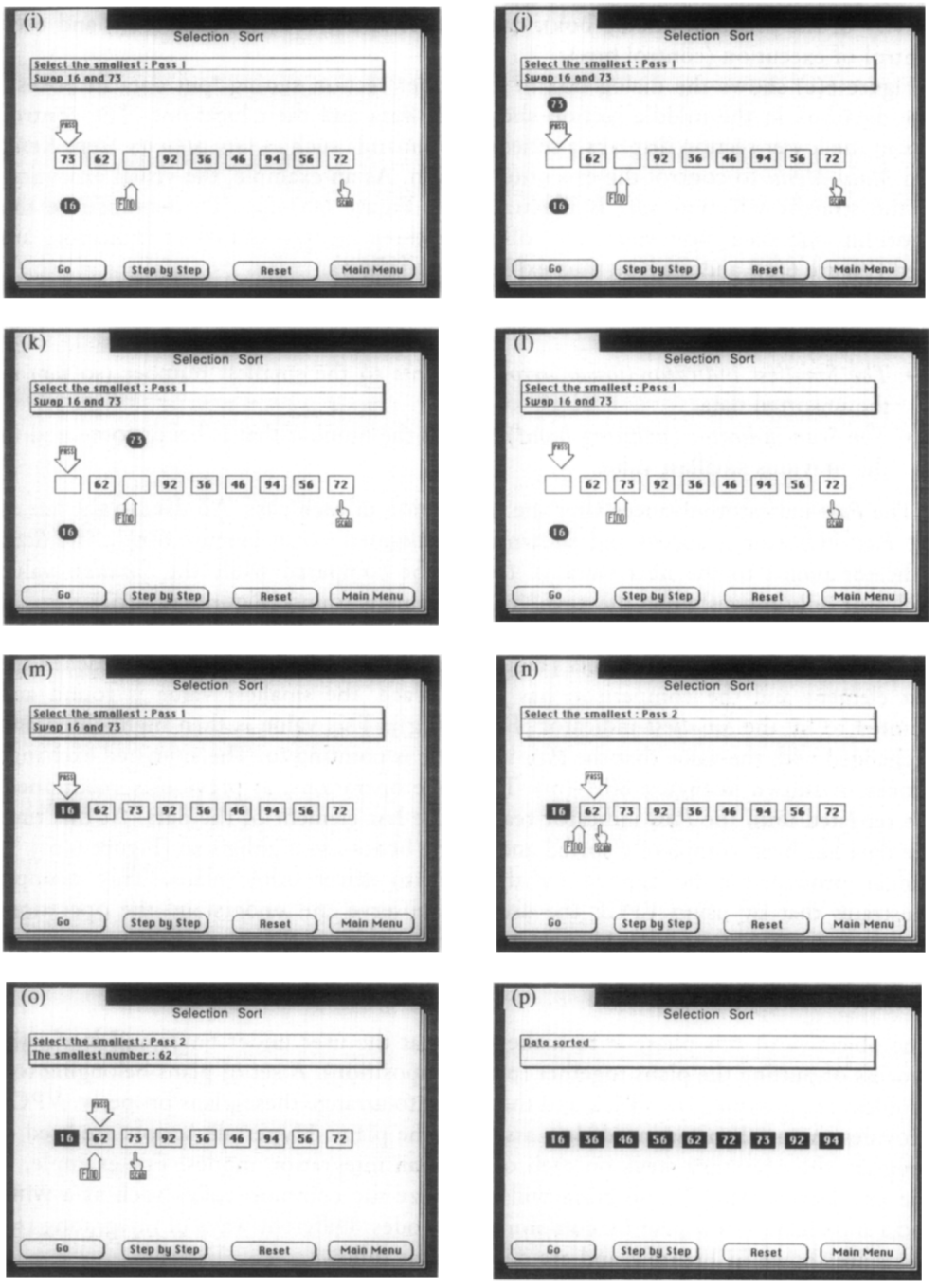

Figure 6. *(Continued)*

4.1.5. *Visual and Descriptive Execution*

VPCL provides the visual execution of the program with data. This includes current activity of the program (dialog box), data update and movement (data box) and user control of execution (control box).

Figure 6(c) shows the dialog box in the upper section stating 'put data in boxes'. The data box in the middle section shows the data and their locations. The control box in the lower section displays a series of commands such as *Go*, *Step by Step*, *Reset* and *Main Menu* to control the execution session. As an example, the visual execution of the straight selection sort is illustrated by Figure 6(a)–(p). To demonstrate the algorithm and data movement, the following three arrows known as indicators are used [Figure 6(c)] and may be thus explained as follows.

- *The Pass indicator* (*upper arrow*). Points to the starting location of the unsorted data.
- *The Smallest indicator* (*lower arrow*). Points to the smallest number (so far) in the unsorted data.
- *The Scan indicator* (*pointing hand*). Shows the number that is being compared to the previous smallest value.

The *Pass* indicator advances after the completion of each pass. All data to the left of the *Pass* indicator is sorted and shown as highlighted boxes [Figure 6(n)]. The *Scan* indicator points to the next value of data to be compared while the *Smallest* value indicator only moves when the next smallest value is found [Figure 6(c)–(o)].

The first execution window shows the initial condition where all of the indicators are pointing to the data elements [Figure 6(c)]. Once the *Scan* indicator reaches the last element and the comparison has taken place, the smallest value is found and pointed to by the *Smallest* indicator [Figure 6(g)]. This value is then highlighted and exchanged with the value that the *Pass* indicator is pointing to. The animated exchange process is shown in Figure 6(h)–(m). The same operations, as previously mentioned, are repeated until the *Pass* indicator reaches the last element of the data. At this time the data has been completely sorted and all the boxes are highlighted [Figure 6(p)]. A similar process can be applied by the user to select other plans. This example illustrates that by using VPCL the user can observe and understand the operations involved in programming problem-solving.

4.2. Phase 2: Plan Integration

The objective of this phase is to make sure that the user understands and learns the process of putting the plans together (plan composition). A set of plans belonging to a problem are provided by VPCL and the task is to arrange these plans properly. VPCL provides the mechanisms needed to assemble the plans. Hypertalk scripts are used to program the necessary steps on each of the plan integration modes. For example, in the interleaved plan, the program will recognize the common codes such as a **while loop** and create a new plan by combining the codes. Different ways of integrating two plans are shown in Figure 7 and are described as follows.

1. *Appended plan.* Plans are integrated one after the other sequentially.
2. *Interleaved plan.* A plan can be interleaved with another plan, in which case either can enter or exit the other.

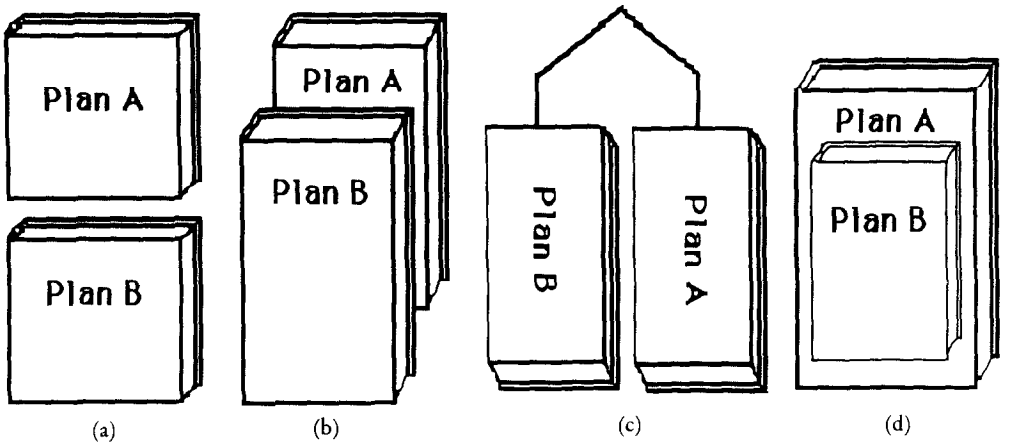

Figure 7. VPCL plan integration mode. (a) Appended plans. (b) Interleaved plan. (c) Branched plan. (d) Embedded plan

3. *Branched plan.* A plan can be diverted to other plans depending on a given condition.
4. *Embedded plan.* A plan can be entirely embedded within another plan.

VPCL facilitates the plan integration so the user can simply click on the selected plans and the desired integration mode. If the plans were incorrectly combined, the system warns the user by sending a message through the dialog box and the user tries again. As the user combines the plans, VPCL generates the program. This process will continue until all of the plans are integrated and the final program is generated. The program will then be executed using similar steps as in phase 1, showing animation of the program during execution.

With regards to learning, plan composition can be applied to the same problem that was just reviewed in the rehearsal phase, a problem different but similar to the problem in phase 1, or a completely new problem. For example, when rehearsing a *Selection Sort* problem in phase 1, the plan composition can be applied to the same *Selection Sort*, the *Exchange Sort* or the *Binary Search*. The *Selection Sort* is similar to the *Exchange Sort* but the *Binary Search* is entirely different.

An example of phase 2 in action starts with Figure 8(a) (showing the plan library). By clicking on the sort plan and phase 2 a window for the sort plan will be displayed. This window shows the various sorting methods that are available [Figure 8(b)]. By clicking on the *Exchange Sort,* VPCL automatically displays a set of plans that are required for this sort [Figure 8(c)]. These plans should be selected and integrated in the proper order.

Selection and integration of plans are shown in Figure 8(d) and (f). For example, exchange plan and compare adjacents plan are integrated by the *Appended* mode [Figure 8(d)], thus creating a new plan. The loop plan and the newly created compare adjacents exchange plan are integrated by the *Embedded* mode [Figure 8(f)]. The new plan will be included in the current plan library for further use. For each of the above plan integration, VPCL displays the programming codes accordingly [Figure 8(e) and (g)]. The complete exchange sort program is shown in Figure 8(h).

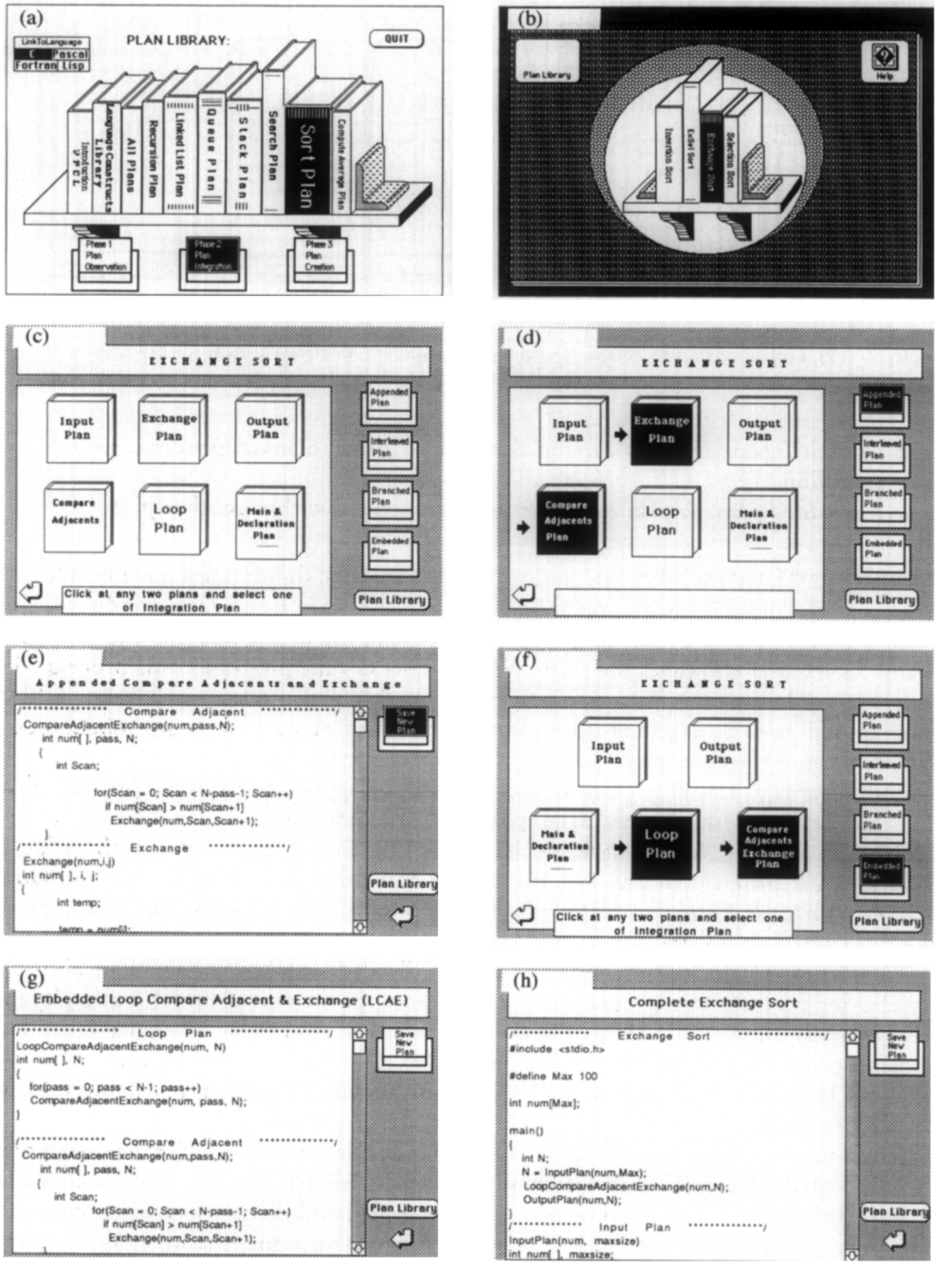

Figure 8. Plan integration phase. (a) and (b) Plan selection. (c) and (d) Plan decomposition and integration. (f) Integration of existing plan with the newly created plan. (e), (g) and (h) Plan code after integration

4.3. Phase 3: Plan Creation

This phase of VPCL deals with the creation of plans and writing actual programming code. To translate a problem into a written program, several plans will be needed. Such plans may be created, may exist or may be modified, thus integrated to form programs. Visual utilities are presented by VPCL for creation of a plan. In this final phase, the user will only be given a statement of the problem. To solve the problem, the user will have to develop all of the steps required to arrive at a solution. These steps include the following.

1. Selection of available plans from the VPCL library, modifications of these plans to meet specific requirements or creation of new plans.
2. Integration of plans and their modification.
3. Execution of the program.

To demonstrate phase 3 of the VPCL in action, a sorting method known as *ExSel Sort* has been chosen. The sort is a combination of the *Exchange* and *Selection* sorts discussed earlier. The user's task is to implement the entire algorithm. Activation of *ExSel Sort* from plan library and description of the sort is displayed in Figure 9(a)–(c). The *ExSel Sort* plan is made of existing available plans such as the *Exchange* and *Selection Sorts* in the plan library. These plans should be loaded and integrated (Figure 9(d). The embedded plan *Integration* mode has been applied on the loop plan with *Compare Adjacents & Exchange* (CAE) as well as *Select Smallest and Exchange* (SSE). This is shown in [Figure 9(d). The programming code for this plan integration, using the C language, is shown in Figure 9(e). These two plans are integrated by interleaving them thus becoming a new plan in the plan library [Figure 9(e) and (f)]. Modifications are made on the resulting plan for accuracy and the complete program is then displayed. This is illustrated in Figure 9(g) and (h).

5. Language Construct Library

VPCL provides a library to help understand the syntax and the semantics of language constructs, related techniques as well as data structures. This library can be used to modify an existing plan or an original plan can be developed. This language constructs library consists of the following.

1. A set of constructs, related techniques and data structures for a designated programming language.
2. Type categorization of each construct.
3. Visualization of constructs and data structures.
4. Examples and tutorial description.
5. A set of keywords and built-in functions.

Some of the features of the language constructs library are illustrated as follows. Figure 10(a) displays the C language constructs library, which was activated through the plan library on Figure 3(a). Selecting the **Loop** constructs displays its type category such as **For, Do..While** and **While**. In addition, visual presentation of each construct category is shown together with the programming language examples. These are illustrated in Figure 10(a)–(f). Alternatively, language construct description and

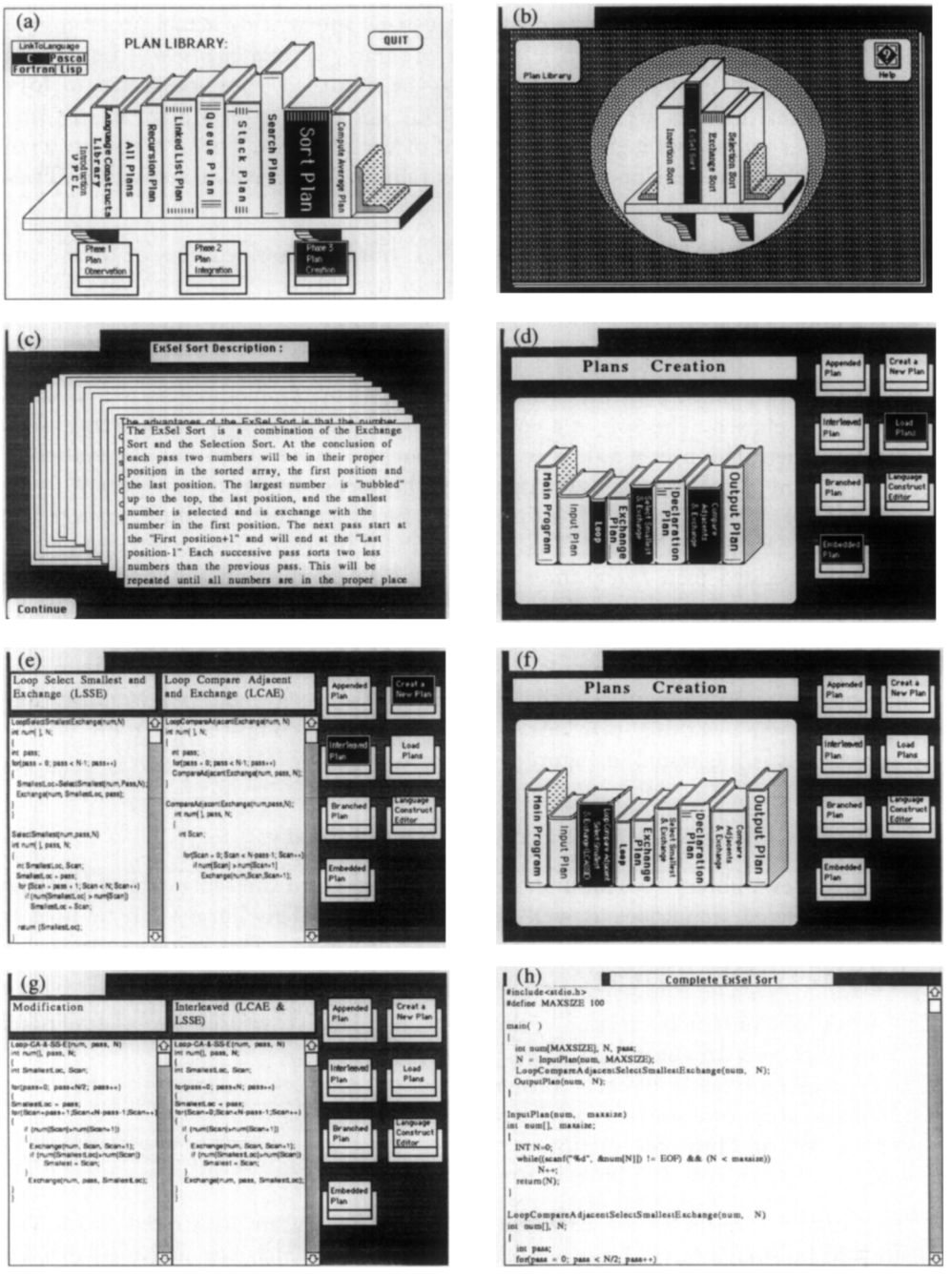

Figure 9. Plan creation phase. (a)–(c) Selection of *ExSel* sort from plan library and its description. (d) Load and integration of plans. (e) and (f) Programming code for plan integration and creation of new plan. (g) and (h) Modification of resulting plans and display of completed program

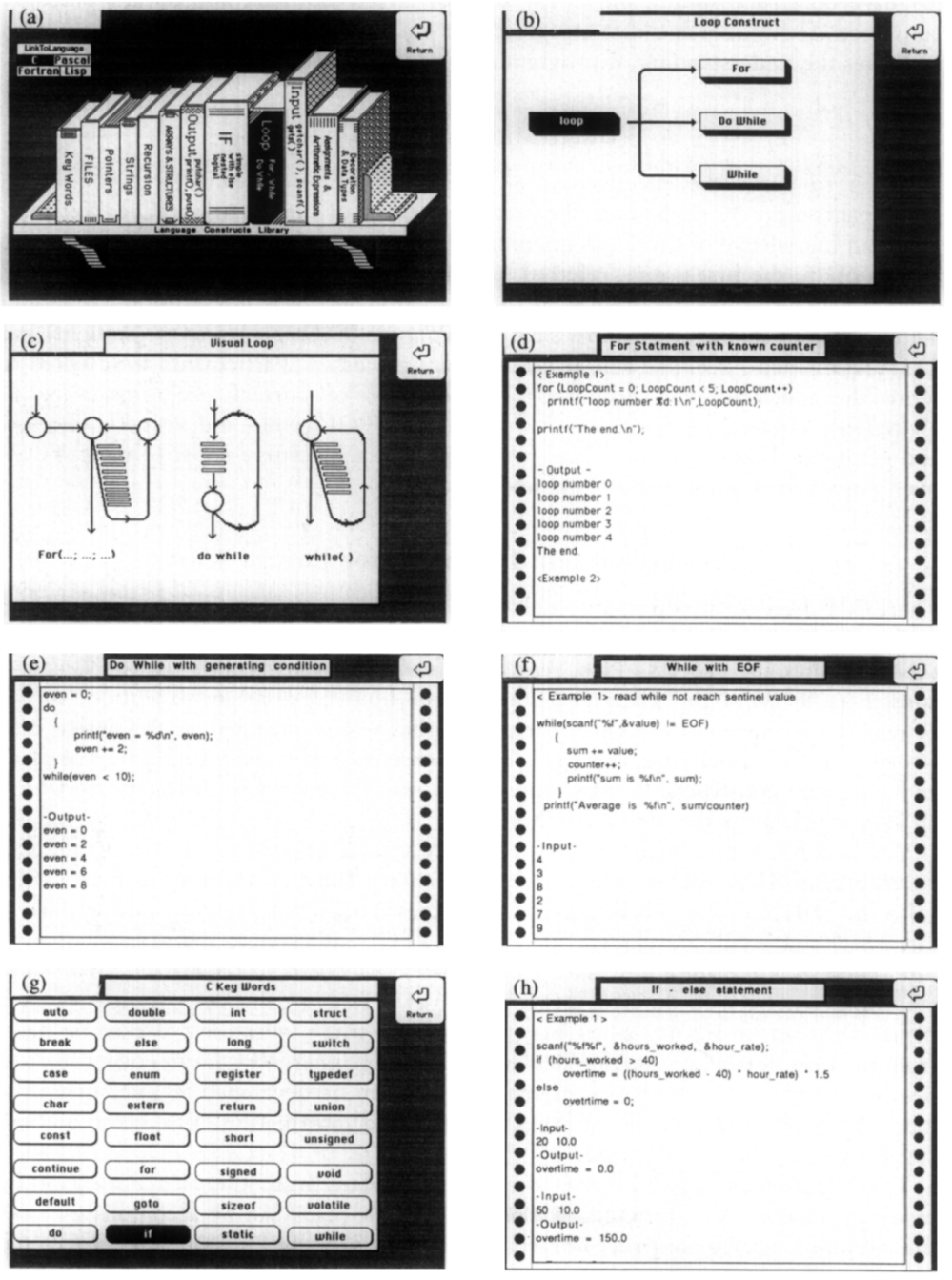

Figure 10. Language constructs library. (a) and (b) Selection of loop construct. (c) Visual representation of loops. (d)–(f) Examples of loops. (g) and (h) Alternative selection of language constructs and example

examples can be chosen through the keywords set [Figure 10(g) and (h)]. This language construct library allows the user to select the most appropriate language constructs, techniques and data structures for a particular problem. This visualization improves the understanding of programming concepts.

6. Visualization Versus Textual

In order to prove the effectiveness of VPCL, an empirical study was conducted. Participants were 30 students at the State University of New York at Old Westbury who had knowledge of the C programming language. Students were divided into two groups of 15; the first group, referred to as the VPCL group, was exposed to VPCL and the second group, referred to as the Textual group, was not exposed to VPCL. Three different sorting algorithms such as *Exchange* (bubble), *Selection* (straight) and *ExSel* (combined *Exchange* and *Selection*) were discussed. Participants were asked to write the indicated sort programs. The number of correct programs were 13 (*Selection*), 11 (*Exchange*) and 10 (*ExSel*) for the VPCL group and, respectively, 9, 7 and 6 for the Textual group. These results suggest the relative superiority of VPCL over conventional programming methods.

7. Conclusions and Directions for Future Work

There is a need to incorporate visualization into current programming languages as well as in the design of future languages. Visualization can be used in several steps of the programming process. Plan representation, plan integration, use of language constructs to build the plans and the control flow of program exceution can all benefit from visualization. All of these steps have been included in the design of a Visual Plan Construct Language (VPCL), which emphasizes the visualization of plans and Language constructs. A plan represents a specific task; it is abstractly shown by an icon which hides the plan's details.

VPCL consists of a plan library with three phases of learning and programming development. These phases range from elementary (plan observation), intermediate (plan integration), to an advanced level (plan creation), where the plans are rehearsed, integrated and finally developed. In phase 2, VPCL indicates to the user whether or not the plan integration for the given problem is correct. To expand upon this, reasoning can be incorporated into phase 3 of VPCL in the development of new plans. This can be accomplished by observing the user's selections of existing plans from the plan library, or creating new ones. In this manner, VPCL can collect clues to determine the user's intent. An inference engine can be developed to implement this kind of reasoning. Based on the clues and gathered information, VPCL can aid and suggest proper courses of action.

VPCL promotes reusability by using existing plans from the plan library or by modifying these plans. In addition, creation of new plans can be accomplished with the aid of a language construct library. This library visually illustrates the syntax and semantics of the constructs, techniques and data structures. VPCL is not language-dependent and can be adjusted to any of the current conventional languages. VPCL enhances the learning of programming since plans are preprogrammed. A great amount of time can be saved in not having to 'reinvent the wheel'. Predefined plans

can be accessed instead of programming them all over again. Detailed grammatical assistance is provided for writing new plans. The obstacles that programmers face, such as syntax, semantics and specific idiosyncrasies of a textual programming language, can be reduced or eliminated, making VPCL a powerful environment for actual programming. This has been shown by an empirical study, which demonstrates the effectiveness of VPCL as a learning and development tool.

References

1. B. Adelson & E. Soloway (1985) The role of domain experience in software design. In *IEEE Transactions on Software Engineering* **SE11**, 1351–1360.
2. B. Shneiderman (1976) Exploratory experiments in programmer behavior. *International Journal of Computer Information Science* **5**, 123–143.
3. R. Brooks (1983) Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies* **18**, 543–554.
4. A. Ebrahimi (1989) Empirical study of errors by novice programmers and design of visual plan construct language (VPCL). PhD dissertation. Department of Computer Science, Polytechnic University of New York.
5. E. Soloway (1986) Learning to program = learning to construct mechanism and explanation. *Communications of the ACM* **29**, 850–858.
6. J. Spohrer & E. Soloway (1986) Novice mistakes: are the folk wisdoms correct? *Communications of the ACM* **29**, 624–632.
7. D. Knuth (1971) An empirical study of FORTRAN programs. *Software—Practice and Experience* **1**, 105–71.
8. P. Bayman & R. Mayer (1983) A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM* **26**, 677–679.
9. J. Dyck (1987) Learning and comprehension of BASIC and natural language computer programming by novices. PhD dissertation. University of California, Santa Barbara.
10. M. Klerer (1984) Experimental study of a two-dimensional language vs FORTRAN for FIRST-COURSE programmers. *International Journal of Man–Machine Studies* **20**, 455–467.
11. N. Cunniff & R. Taylor (1987) Graphical vs. textual representation: an empirical study of novices' program comprehension. In: *Empirical Studies of Programmers*: *Second Workshop* (G. M. Olson, S. Sheppard & E. Soloway, eds) Ablex, New Jersey.
12. C. Yu & S. Robertson (1988) Plan-based representation of Pascal and FORTRAN Code. In: *CHI'88 Conference Proceedings* (D. Frye & S. Sheppard, eds.) Washington D.C.
13. J. Bonar & B. Liffick (1990) A visual programming language for novices. In: *Visual Programming System* (S. Chang, ed.) Prentice Hall, New Jersey.