Int. J. Human–Computer Studies (1994) **41**, 457–480

Novice programmer errors: language constructs and plan composition

ALIREZA EBRAHIMI

Department of Computer Science, State University of New York, College at Old Westbury, Old Westbury, NY 11568, USA

(Received 15 July 1991 and accepted in revised form 23 August 1994)

Why do novice programmers have difficulties in programming, and what are the probable causes of these errors? This study analyses the role of Language Constructs comprehension, Plan Composition, and their relationship to each other as applied to novice programming errors. The experiment was conducted with 80 novice programmers who were divided into four groups of 20. Each of the groups enrolled in one of the following programming language courses: Pascal, C, FORTRAN, or LISP.

The results of the study indicate that the misunderstanding of Plan Composition and semantic misinterpretation of Language Constructs are the two major causes of errors. In addition, the study has concluded that these errors are highly correlated.

1. Introduction

Writing error-free programs has been one of the primary goals of programmers in the computer age. A variety of techniques have been advocated to achieve this objective. Some of these methods include flowcharting, structure and defensive programming, verification of the program for correctness, functional, logical, and object-oriented programming. Despite the availability of these techniques, programmers still find it necessary to spend a considerable amount of time rectifying their errors.

Several studies have been conducted on novice programmers to understand the causes of errors and why they are committed (Weinberg, 1971; Weissman, 1974; Anderson & Jeffries, 1985; Bonar, 1985; Johnson, 1986; Spohrer, Soloway & Pope, 1985; Soloway, 1986; Eisenberg, 1987; Yu & Robertson, 1988; Detienne & Soloway, 1990).

The study of novice programming errors can lead to a better understanding of problem-solving strategies and will highlight the difficult aspects of programming and programming instruction. Furthermore, it will contribute to the refinement of programming languages, training tools and design methods.

The significant features of this study include an investigation of novice programmer errors by conducting two separate experiments, one in Language Constructs and the other in Plan Composition. In addition, the relationships between Language Constructs and Plan Composition errors of novice programmmers are analysed. Another aspect of this study is the use of four programming languages—Pascal, FORTRAN, C, and LISP—instead of only one or two languages. In conjunction with each experiment, each novice programmer was interviewed in order to clarify the causes of his/her errors. The study further investigates different types of errors in each language, but does not analyse the relationship across the languages with

1071-5819/94/010457 + 24$08.00/0

respect to the number of errors committed. This would be an interesting subject for future investigation.

2. Language constructs comprehension

Language Constructs are important tools for programming. It is quite evident that novice programmers have some misconceptions about Language Constructs (Bayman & Mayer, 1983). Some of the questions in regard to the syntax and semantics of Language Constructs include: What are the differences among the loops? What are the default rules of a language? How is the control variable of a "for" loop undated? how is a value bound to its variable? How do the nested "if" and nested loops work? Novice programmers tend to transfer natural language expressions into a programming language.

Different programming languages evoke different programming styles, thus generating their own type of errors. As a result, the type and number of errors vary from language to language (Knuth, 1971; Klerer, 1984; Ebrahimi, 1989). For example, in languages like C and FORTRAN the resulting value of "5/9" is zero, since 5 and 9 are both integers. Furthermore, Pascal has two division operators, one for integer division ("div") and one for real division ("/"). These notations are hindrances to cognitive comprehension and conflict with conventional mathematical notations.

The following examples demonstrate some of the difficulties encountered because of a particular programming language design.

2.1. PASCAL READ STATEMENT

Since input handling in Pascal is format-free, several problems will arise when the data needs to be read in a certain manner. For instance, if the data has the form "`33 M 3500.50`", standing for AGE, GENDER and SALARY, an extra step is needed to read the data correctly.

A "dummy" variable is required to read the existing blank space between AGE and GENDER.

i.e. `Read(AGE, BLANK, GENDER, SALARY)`

An alternative solution would be to remove the blanks between the number and the character representing AGE and GENDER.

i.e. `33M 3500.50`

2.2. C EMBEDDED EXPRESSION, EQUALITY OPERATOR, AND VARIABLE NAME (UNDERSCORE)

The use of embedded expressions and the substitution of the equality operator "=" with "==" has caused much confusion. For example, in a statement such as `while(s[i]=t[i])`, where "`s`" and "`t`" are strings, someone who is not very familiar with C will conclude that the two strings are compared in the loop. This is not the case, however, since "`=`" is an assignment operator, not the equality operator. The loop will assign every value of "`t`" to "`s`" and will terminate when the string "`s`" reaches null (zero), since the last character of a string in C is always a null.

Another example of confusion would be the use of "_" (underscore) as an identifier. A statement such as "- - _" is a valid statement, which decreases the value of variable "_" by 1.

2.3. FORTRAN DEFAULT TYPE CONVERSION

In FORTRAN any variable that starts with the letter "`I`" through "`N`" is considered an integer. Therefore, the assignment of a decimal number to an integer variable will result in a truncation of the integer. For example, in the statement `NUM=4.5`, the value 4 will be assigned to the variable `NUM` instead of the decimal value 4.5.

2.4. LISP DO LOOP

In most dialects of LISP, such as Scheme and Common Lisp, there is a misconception concerning the execution of the Do loop because of its sequential structure.

i.e. `(do ((i 0 (+ 1 i)) (sum 0 (+ sum i))) ((=? i 10) sum))`

The above program enables one to compute the sum of the first ten natural numbers. The variables "`i`" and "`sum`" will be set to an initial value of zero. The updates are represented before the condition of loop but will not be executed until the test has been done.

3. Plan composition understanding

Plans are chunks of meaningful information (canned solutions) for reaching the problem solution (Soloway, 1986). Psychologists and artificial intelligence researchers have employed the concept of Plan in order to understand knowledge systems, i.e. how concepts are structured and developed in the human mind, and how they are used in understanding behavior (Schank & Abelson, 1987).

The study of human problem solving shows that template-like solutions, or Plans, are used in solving a problem. In programming, a Plan consists of related pieces of code representing a specific action. The Plan is implemented using Language Constructs and can be as short as a single statement (i.e. "`i++`" in C) or as long as an entire group of codes which may itself contain other plans, e.g. Sort Plan (Ebrahimi, 1992).

In a study of novice programmers, we want to know how the Plans are implemented and integrated with each other. Novices have difficulties managing the

Plans and thus they produce errors. Most errors seem to arise as the novice tries to put the "pieces" of a program together in the Plan Composition (Soloway, 1986; Spohrer & Soloway, 1986*a*, *b*).

For example, a novice may not know how to merge a Plan which checks for an invalid data entry and a Plan which loops through the data. A Plan identifies a particular task, such as a programming problem, which itself may consist of other Plans. For the sake of an explanation, a Plan would be analogous to a theater play, which itself contains other Plans such as actors and scenes.

The programming errors are related to the mismanagement of Plans and the programming knowledge being used. Therefore, error classifications are based on Plan differentiation, i.e. the distinctions between the Plans that the programmer intended to use, or should have used, and the actual code (Johnson & Soloway, 1983). The following example illustrates the steps involved in solving an elementary problem using the Plan approach, which is similar to a previous study known as the "rainfall problem" (Soloway, 1986).

Example: Find the average of a data set terminated by a sentinel value 9999.
The above problem can be divided into the following subtasks:

1. Input a series of numbers.
2. Find the average.
3. Output the average.

The following plans, which are similar to Soloway's group, are used to map each task and to build the Average program:

1. Read the numbers (ReadAllNumber Plan).
2. Count the numbers (CountAllNumber Plan).
3. Add the numbers (SumAllNumber Plan).
4. Check division by zero (Guard Plan).
5. Compute the average (Average Plan).
6. Print the average (Output Plan).
7. Print error message (Error Plan).

Figure 1 shows a visual representation of the Plans using Pascal.

3.1. PUTTING THE PLANS TOGETHER (PLAN COMPOSITION)

The following three methods are used to integrate the Plans. The visual representation of the Plan integration is shown in Figure 2:

1. **Appended Plans:** One Plan is immediately followed by another Plan (Figure 2(*a*)).
2. **Interleaved Plans:** A Plan enters into and exits from another Plan. In Figure 2(*b*), after activation of some code in Plan A, Plan B will be activated; similarly, after activation of some code in Plan B, Plan A is activated. Interleaving Plans share some common code.

(a) Read All Number Plan

```
read(number);
while(number <> sentinel)do
begin
.
.
.
read(number)
end;
```

(b) Count All Number Plan

```
count := 0;
while(number < > sentinel)do
begin
.
.
.
sum := sum +1
end;
```

(c) Sum All Number Plan

```
sum : = 0;
while (number<>sentinel)do
begin
.
.
.
sum : = sum + number
end;
```

(d) Guard Plan (division by zero)

```
if count <> 0 then
begin
...
end
else
```

(e) Average Plan

```
average : = sum/count;
```

(f) Output Plan

```
write In (Average is, avg);
```

(g) Error Message Plan

```
write In ('Error in data')
```

FIGURE 1. Visual representation of Average program Plan.

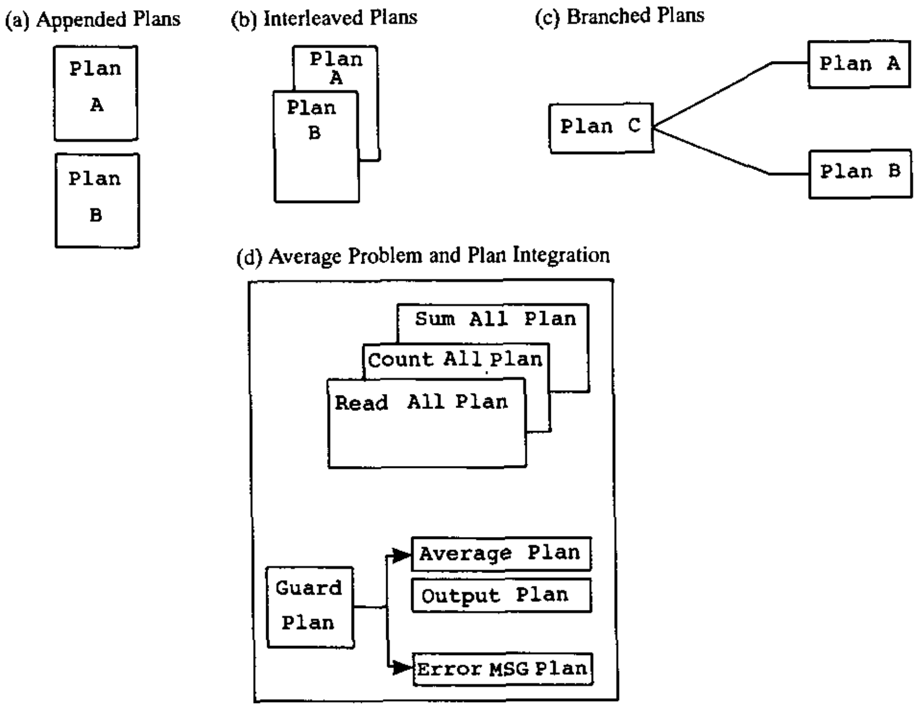

FIGURE 2. Pictorial representation of Plan integration.

(a) Interleaved Plan : Read All Number Count & Sum All Number (RCS)

```
sum: = 0:
count := 0;
read (number)
while(number <>sentinel) do
begin
sum := sum + number;
count : = count + 1;
read (number)
end;
```

(b) Appended Plan : Average & Output

```
average : = sum/count;
write In('Average is',average);
```

(c) Branched Plan : Average Output Plan & Error MSG Plan (AOE)

```
if count <> 0 then
begin
average : = sum/count;
write In ('Average is',average);
end
else
write In ('Error in data')
```

(d) Appended RCS & AOE

```
sum: =0:count :=0;
read (number)
while (number <> sentinel) do
begin
  sum := sum + number;
  count : = count + 1;
  read (number)
end;
if count <> 0 then
begin
  average : = sum/count;
  write In('Averageis',average);
end
else write In ('Error in data')
end
```

(e) Complete Average Problem Plan

```
program Find Avg (input,output);
var number,sum,count,integer;
average : real;
begin
  sum: =0:
  count :=0;
  read (number)
  while (number <> sentinel) do
  begin
   sum := sum + number;
   count : = count + 1;
   read (number)
  end;
  if count <> 0 then
  begin
  average :sum/count;
  write In('Avg is',average:5:2)
  end
  else
  write In ('Error in data')
end
```

FIGURE 3. Plan code integration: average problem.

3. **Branched Plans:** A Plan is selected based on the results of a test condition (Figure 2(*c*)).

3.2. PLAN COMPOSITION EXAMPLE: AVERAGE PROBLEM

A pictorial representation of Plan integration for the Average problem is shown in Figure 2(*d*). Figure 3 illustrates the integrated Plan codes for this problem. A brief description of the integrated Plans are as follows;

1. **Interleaved Plans:** ReadAllNumber, CountAllNumber, and SumAll Number Plans are interleaved (RCS) (Figure 3(*a*)).
2. **Appended Plans:** Average Plan and Output Plan are appended (AO) (Figure 3(*b*)).
3. **Branched Plans:** AO Plans and an Error Message Plan are branched (AOE) (Figure 3(*c*)).
4. **Appended Plans:** RCS and AOE are appended to make the final program (Figure 3(*d*)). A declaration and a header are appended.

3.3. LANGUAGES AND PLAN

Language Constructs are building blocks for implementing the Plan. Different programming languages may use similar constructs to implement the Plan. However, a language may emphasize its own particular data structures and features, such as recursion in LISP and pointers in C. As a result, the structure of the Plan will change as the programmers learn the use of the special features of the language.

```
(a)
 main()
 {
   int number;
   float sum, count;
   for (count=0, sum=0; scanf(''%d'',&number); sum+=number, ++count);
   (count?printf(''%f'',sum/count):printf(''divided by zero'');
(b)
  (define (FindAverage)
   (do ((count0(+1 count))
   (sum 0 (+ sum num))
   ((not (number? (set! num(read)))) (if (zero? count) 'division by zero
   (/ sum count)))))
(c)
  (define (Average lst) (if(null? lst)'division-by-zero (/ (sum lst) (count
  lst))))
   (define (sum lst) (if (null? lst)0 (+ (car lst) (sum (cdr lst)))))
   (define (count lst) (if(null? lst)0 (+ 1 (count (cdr lst)))))
```

FIGURE 4. Average program in C and LISP using iteration and recursion.

The previous average program can be written differently, in C and LISP, as shown in Figure 4.

In the C program (Figure 4(*a*)) three Plans, *ReadAllNumber, SumAllNumber* and *CountAllNumber,* are combined in the For loop and written in one line. This is one of the C language features which allows multiple Plans to be implemented differently. In the LISP programs, the programmer uses the concepts of both iteration (Figure 4(*b*)) and recursion (Figure 4(*c*)). Recursion is emphasized more in LISP than in other languages.

4. Methodology

This study investigates the causes of errors based on both Language Constructs and Plan Composition, and their correlation.

The data was collected from 80 undergraduate students enrolled in computer courses at State University of New York, College at Old Westbury. Four groups, consisting of 20 students, each attended one of the following courses: Programming I—Pascal (CS3510); Unix & C (CS3530); FORTRAN (CS3520); and AI Programming—LISP (CS3540). Two experiments, Language Constructs and Plan Composition, were conducted on each group.

For the Language Constructs experiment, a series of small programs, known as segments, were given to students to test the understanding of Language Constructs. For the Plan Composition experiment the students were asked to write a program known as "rainfall". The experiments were conducted during the 6th week of the semester for the Pascal class and during the 4th week of the semester for the C, LISP and FORTRAN classes. The extra time was given to the Pascal group to "even out" the gap, since the other groups had some knowledge of programming.

Each student was requested to sumbit the final working version of the rainfall problem and the results of the program segments. The C and FORTRAN programs were written under the Unix operating system running on a Microvax, whereas the Pascal and the LISP (Scheme) programs were written using the Apple Macintosh. After the completion of the experiments, an interview of approximately fifteen minutes was conducted with each student.

The purpose of these interviews was to identify the probable causes of each student's errors. In these interviews, students were asked to "think aloud" as they went through their programs and solutions of the respective segments.

5. Language constructs experiment

In the Language Constructs experiment, the following categories of segments were used.

1. **Input/Output**
2. **Assignment statements**
3. **If statement variations**
   (a) If without else
   (b) If with else
   (c) Nested if
   (d) Logical if
   (e) Compound if
4. **Loop variations**
   (a) While
   (b) Repeat until (do while)
   (c) For

Additional considerations were given to the most frequent Language Constructs used by the novice programmer and to special features of the language.

The following segments illustrate a portion of the Language Constructs experiment in Pascal and C. The complete Language Constructs experiments are shown in Appendix A for each language.

(a) Run the following program segment separately each time with the data given for the value of *n*:
   0 −5 5.

```
if n>= then
  if n=0 then writeln (''Here now'')
  else writeln (''not here'')
else writeln (''no where'');
```

(b) Run the following program segment separately each time with the given data sets:
   6 2 −1 9999
   9999 −1 2 6

```
scanf(''%d'',&n);
```

```
while(n!=9999)
{
  printf(''%d'',n);
  scanf(''%d'',&n);
}
```

The students were requested to run each program segment mentally with given data, without the use of a computer, and to record the expected output. Specific mistakes each student made were noted. At the conclusion of this part of the experiment, each student was asked to process the segments while "thinking aloud". The purpose of this additional step was to identify and classify the errors more accurately (Ericsson & Simon, 1983). Consideration was given to minimize the novice's conceptual inferences, such as non-intuitive variable names. For example, the statement "s := s+n" was used rather than "sum := sum+num". In the latter assignment, the novice may cognitively infer that the program is trying to add a sequence of numbers. Other assumptions, such as an initialization, an input or a loop, can also be inferred.

A categorization scheme was developed to classify the type of Language Constructs errors committed by each novice programmer. This is shown in Table 1. Some of the Pascal Language Constructs error classifications include: Input, Output, Assignment, IF statements (simple, nested, logical and compound), and Loops (While, Repeat, and For).

An entry of "+" in the table indicates the type of error committed by a student. Such an entry shows that the student had difficulty understanding the specific construct. For example, subject 12 (Pascal) made the following error types: Input, Logical IF, Compound IF, and While loop.

5.1. LANGUAGE CONSTRUCTS ERRORS

Figure 5 shows the Language Constructs error types for the four languages. In Pascal and C the highest percentage of errors occurred in the use of IF statements. In FORTRAN the most errors were made in assignment. The most common errors made in LISP were in the use of logical operators. The most common errors in language constructs are as follows.

IF statements

This type of error was the most common over all the languages. Many of the errors occurred in a logical expression within an IF statement. For example, many students did not understand negation rules for Boolean expressions containing AND and OR. In C, students had trouble knowing when to use the assignment ("=") operator as opposed to the equal ("==") operator. The same problem occurred with both the logical OR ("||") verses the bitwise OR ("|") and the logical AND ("&&") verses the bitwise AND ("&").

In Pascal, students had trouble with grouping statements after IF and ELSE with BEGIN and END constructs. Students had difficulty following the control flow of nested IF statement and matching ELSE with the proper IF statement. Since LISP

TABLE 1
Language Construct error type

(a) Pascal

				IF					LOOP			
Subject ID	Input	Output	Assign	Without	With else	Nested	Log oper	Comp	While	Repeat	For	Sum
1						+	+		+			3
2							+	+		+		3
3	+						+		+	+	+	5
4	+			+				+			+	4
5							+		+	+		3
6	+	+	+		+				+		+	6
7	+	+				+	+			+		5
8	+	+		+		+	+			+	+	7
9	+		+	+	+	+	+	+	+	+	+	10
10	+			+		+	+	+		+	+	7
11				+	+		+	+		+	+	6
12	+						+	+	+			4
13												0
14								+	+			2
15	+			+		+	+	+	+	+		7
16	+											1
17							+					1
18	+			+		+	+	+	+	+	+	8
19		+				+	+	+		+	+	6
20	+		+		+	+	+	+	+	+		8
Sum	12	4	3	7	4	9	15	11	10	12	9	96

(b) C

				IF					LOOP			
Subject ID	Input	Output	Assign	Without & =	With else	Nested	Log oper	Comp if	For	While	Do	Sum
1							+					1
2	+	+		+					+			4
3				+							+	2
4	+			+			+				+	4
5									+			1
6			+				+			+	+	4
7		+		+				+	+		+	5

8		+										1
9	+											1
10		+		+			+	+		+	+	6
11			+	+			+	+	+		+	6
12												0
13		+		+		+	+				+	5
14	+	+		+			+		+			5
15				+	+		+	+			+	5
16				+								1
17	+	+	+	+	+	+			+		+	8
18							+					1
19												0
20	+	+	+	+		+	+				+	7
Sum	6	8	4	12	2	3	10	4	6	2	10	67

(c) FORTRAN

Subject ID	Input	Output	Assign & Conver	IF					LOOP			Sum
				Without	With else	Nested	Log oper	Comp	While	Repeat	Do	
1	+										+	2
2				+		+	+		+	+	+	6
3		+	+		+	+	+	+	+	+	+	9
4	+		+	+		+						4
5	+		+					+				3
6	+	+	+		+	+	+			+	+	8
7						+		+			+	3
8												0
9	+		+	+	+	+	+	+	+	+	+	10
10			+								+	2
11	+		+		+	+	+					5
12			+			+	+	+		+	+	6
13	+											1
14			+	+			+					3
15		+	+				+		+		+	5
16	+											1
17		+	+									2
18	+	+	+		+	+	+					6
19	+					+	+		+			4
20							+					1
Sum	10	5	12	4	5	10	11	5	5	5	9	81

TABLE 1 (*Continued*).

(d) LISP				CONDITIONS					LOOP					
Subject ID	Input	Output	Assign	Without else	With else	Nested	Log oper	Comp	Do	While	Sum1	CAR CDR Cons	Recur	Sum
1											0			0
2	+						+		+		3	+	+	5
3							+	+			2		+	3
4						+			+		2		+	3
5								+			1			1
6							+	+			2			2
7	+		+		+	+	+	+	+		7	+	+	9
8		+					+		+		3		+	4
9						+				+	2			2
10								+	+		2	+	+	4
11											0	+	+	2
12									+		1			1
13											0			0
14	+			+	+		+	+	+		6	+	+	8
15		+			+	+	+			+	5	+	+	7
16	+	+					+	+			4			4
17						+	+		+		3		+	4
18	+		+				+	+		+	5	+	+	7
19							+	+	+		3			3
20	+			+		+	+	+	+		6	+	+	8
Sum	6	3	2	2	3	6	12	10	10	3	57	8	12	77

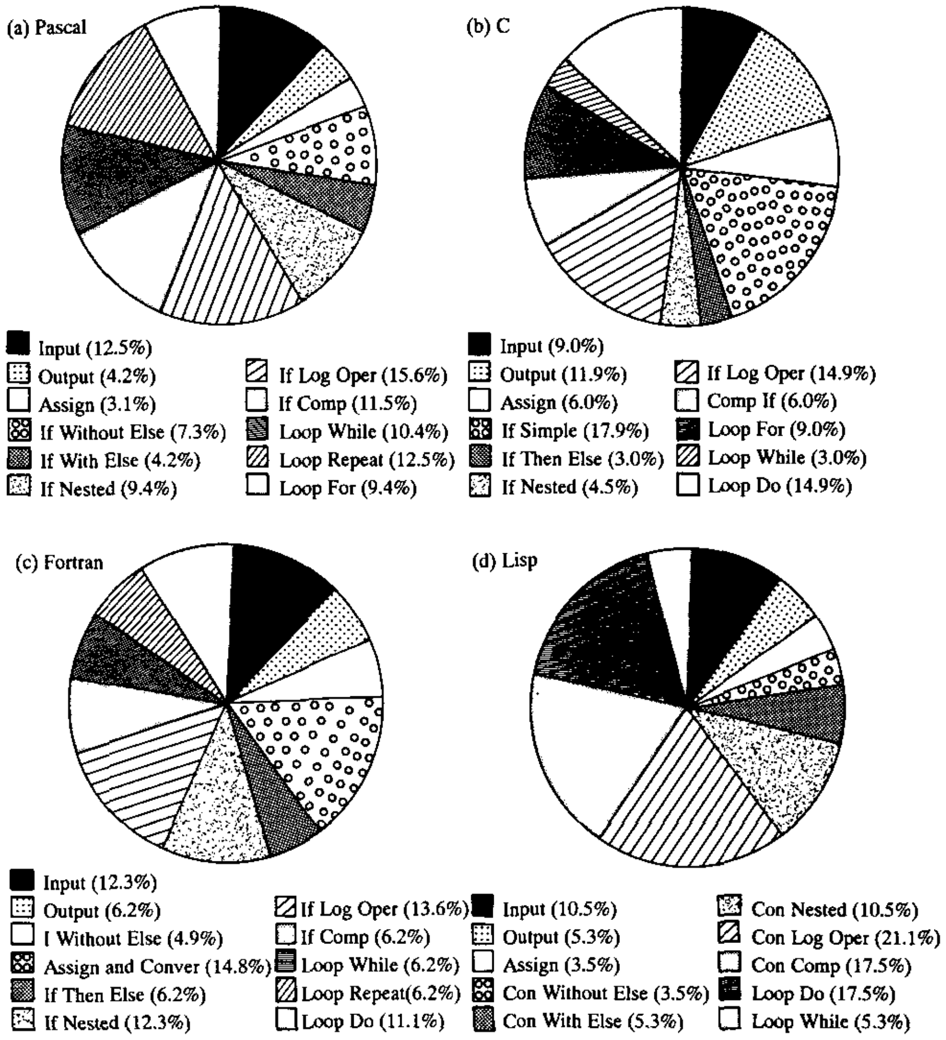

FIGURE 5. Language Constructs error types.

provides two major forms of a conditional statement, IF and COND, the different syntax and semantics of these constructs has led to major confusion.

Loops

Students had trouble with how and when to terminate loops. For example, the DO loop and the REPEAT loop are very different. Students did not notice the difference easily and treated them like a WHILE loop, thinking the condition will always be checked before any statement is executed. In C, the FOR loop caused a problem because it does not clearly correspond to its evaluation and execution. The complexity of the DO loop structure in LISP made it difficult for students to understand.

Input/output
Students had difficulty with the format of input and output. For example, each language provides a type of read statement, requiring some degree of formatting. This was beyond the comprehension of most students.

In C, the PRINTF function did not provide a default format. Therefore, if none is specified, the output will not be printed. Because Pascal does not provide input formatting, students had trouble with the READ and READLN statements. FORTRAN can read and print data with or without format, but students did not understand the real concept behind the format.

Assignment
Most of the errors with assignment occurred in FORTRAN because of the way FORTRAN treats variables. Variables starting with "I" through "N" are automatically treated as integers, and thus there will be truncation of the decimal part. For example, the netpay = 1000.99 leads to the assignment of 1000 to netpay, because netpay is automatically an integer, not a real number. The expression C = 5/9 * (F-32.0) always results in 0, because integer division (5/9) is 0.

6. Plan composition experiment

The problem used for the Plan Composition experiment is known as "rainfall". This problem has been used extensively in previous empirical studies (Johnson, 1986; Bonar & Cunningham, 1988; Frye, Littman & Soloway, 1988). The rainfall problem has the following characteristics:

(a) Several Plans such as Sum, Count, Average and Maximum are used.
(b) A variety of Language Constructs such as input, output, assignment, IF statement and loops are used.

"Rainfall" problem: Write a program that will read the amount of rainfall for each day. A negative value of rainfall should be rejected, since this is invalid and inadmissible. The program should print out the number of valid recorded days, the number of rainy days, the rainfall average over the period, and the maximum amount of rain that fell on any one day. Use a sentinel value of 9999 to terminate the program.

The submitted programs were evaluated and errors were recorded by the author who taught the courses. These errors were classified based on the difference between the correct Plan and the intended Plan used by the novice programmer.

This method is similar to the error classification used by Johnson and Soloway (1983) for novice programmers and by Ostrand and Weyuker (1984) for professional programmers. The errors were classified into four major categories which had six sub-categories. This type of classification clarifies any discrepancy that might arise between errors. The four major categories used for Plan differences are:

1. **Missing:** An entire Plan or one or more of its components are missing, e.g. sum initialization is missing.
2. **Misplaced:** An entire Plan or one or more of its components appears in the wrong place, e.g. counter initialization done inside the loop.

3. **Malformed:** An entire Plan or some of its components is formulated incorrectly, or is only partially correct, e.g. "`num := sum+num`" instead of "`sum := sum+num`".
4. **Misused (Spurious):** An irrelevant Plan, or component thereof, is used, e.g. using a counter for counting the negative numbers.

Each of the above Plan difference categories was further divided into sub-classifications known as Plan components, for example, "Missing initialization" where "missing" is the Plan difference and "initialization" is the component. The Plan component sub-classifications are the following:

1. **Initialization:** Initialization of a variable, e.g. `sum := 0`.
2. **Input:** Reading the input data, e.g. `READ(num)`.
3. **Output:** Output the data, e.g. `WRITE(num)`.
4. **Update:** Change the value of a variable by new assignment, e.g. `counter := counter+1`.
5. **Guard:** Test condition, e.g.
   (a) If: `If num<0` .... (b) loop: `While(not eof)`.....
6. **Declaration:** Data characteristics such as variable, type and constand declaration, e.g `int average`.

6.1. PLAN ERROR TABLE

A Plan Error Table summarizes the error classifications of novice programmers based on Plan differences (rows) and Plan components (columns). Each entry in the table indicates the number of errors for each category. Table 2 illustrates a Plan error, with each entry specifying a sample error. The column sub-heading, Loop, refers to errors involving the control variables(s) of the loop. The Non-loop column subheading refers to errors involving variables that do not belong to the loop.

6.2. WHAT IS CONSIDERED AN ERROR?

In this study, errors are not simply limited to missing code or malformed statements, but also include any unnecessary or misplaced code even if the output has been achieved. Novice programmers often think that, since there are no syntax or run-time errors, their programs are correct. Their argument is "My program ran, so therefore it must be correct." They tend not to distinguish among errors in different phases of programming, such as compile time, run time, and logic. Moreover, the novice programmer usually does not consider efficiency to be important. The following are some examples of these errors:

1. **Code Misplacement:** Plan component is incorrectly placed. For example, computing the average value inside a loop:
```
while not eof do
  begin
  . . .
  avg := sum/count
end
```
2. **Spurious Initialization:** Unnecessary initialization of a variable. For example, the initialization of a variable before it is read or assigned:

TABLE 2
Plan error classification with sample entries

	Plan components									
			Init		Update		Guard			
Plan differences	Input	Output	Loop	Non-loop	Loop	Non-loop	Loop	if case	Syntax concept	Others
Missing	read	write	i := 1	sum := 0	i := i + 1	s := s + a	while note of	if max > num	begin end	var declare
Misplaced	read outside loop	write inside loop	i := 1 in loop	sum := 0 inside loop	i := i + 1 outside loop	average in loop	sentinel and repeat	if num < 0 before read num	begin end	
Malformed	readln for read	writeln	i := 0	sum := −9999	− := i + num	avg := sum div n	> for >=	< = 0 for <	else	avg as integer
Spurious & misconception	extra read	extra write	j := 1	avg := 0	j := j + 1	sum := sum + avg	extra loop	extra if	if for while	extra var

```
rain := 0
avg := 0;
read(rain);
avg := sum/count;
```

3. **Missing Initialization:** Novices may neglect to initialize a variable because they assume that all variables used in a program are automatically initialized to zero by the system.
4. **Spurious Coding:** Additional unnecessary code is used in the program but not required by the original specification, e.g. counting the negative values or printing the sums when they are not required.

6.3. PLAN COMPOSITION ERRORS

The most common errors in Plan Composition are as follows.

Guard IF
This feature was the most common source of error among all four languages. Students neglected to take into account, when using IF statements, the need to check for special situations, such as division by zero. Understanding where to place the guard IF was another problem. This situation indicates that textbooks and/or instructors fail to emphasize error checking.

Initialization
This was another major source of Plan Composition errors. Students frequently overlooked the initialization of variables because of the assumption of automatic initialization by compilers. Some Pascal compilers initialized variables to zero although it is not specifically defined by the language.

Update
Students had problems with both improper and unnecessary updating of variables. For example, some students updated the average inside the loop, which works, but is unnecessary. Also, the updating of counter variables was a problem.

Loops
Students had difficulty with what type of loop to use, how to terminate the loop, and the structure of the loop.

Eighty programs were analysed within the Plan Composition experiment. The subjects for this test were the same as those in the Language Constructs experiment. Table 3 tabulates the Plan-component errors for each of the four Plan-difference categories. Table 4 shows the number of errors for each student. Figure 6 represents the errors for the Plan differences. A total of 125, 80, 108 and 81 Plan errors were observed for Pascal, C, FORTRAN and LISP respectively. The results of the Plan Composition experiment show that students have difficulty putting Plans together.

TABLE 3
Plan error

	Plan components							
					Guard			
Plan differences	Input	Output	INIT	Update	Loop	If	Declare	Sum
(a) Pascal								
Missing		2	12	1		22		37
Misplaced	1	2	1	16	1	3		24
Malformed		3		5	14	7	2	31
Spurious	3	4	11	3	2	4	6	33
Sum	4	11	24	25	17	36	8	125
(b) C								
Missing		4	3	2		16		25
Misplaced	1	1		5		2		9
Malformed	3		4		6	2	3	18
Spurious	2		16	4	1	2	3	28
Sum	6	5	23	11	7	22	6	80
(c) FORTRAN								
Missing		3	14	12		19		48
Misplaced	1			8		1		10
Malformed	1	3	2	7	5	3	6	27
Spurious	2	9	4	4	1	2	1	23
Sum	4	15	20	31	6	25	7	108
(d) LISP								
Missing		5	1	4		10	—	20
Misplaced	1	1	1	8		4	—	15
Malformed	1	2	4	8	5	6	—	26
Spurious	1	8	2	2	6	1	—	20
Sum	3	16	8	22	11	21	—	81

TABLE 4
Student plan error

Subject ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
(a) Pascal no. of errors	4	5	7	6	4	8	7	10	15	7	6	4	1	2	9	2	0	11	8	9
(b) C no. of errors	3	5	3	3	0	3	6	0	2	8	8	0	5	5	4	4	10	2	0	9
(c) FORTRAN no. of errors	2	10	12	5	4	10	4	0	15	0	8	10	0	3	7	0	2	9	5	2
(d) LISP no. of errors	0	6	4	3	0	4	8	6	2	4	0	1	1	8	6	6	5	6	3	8

Please note that subject IDs represent different subjects for each language.

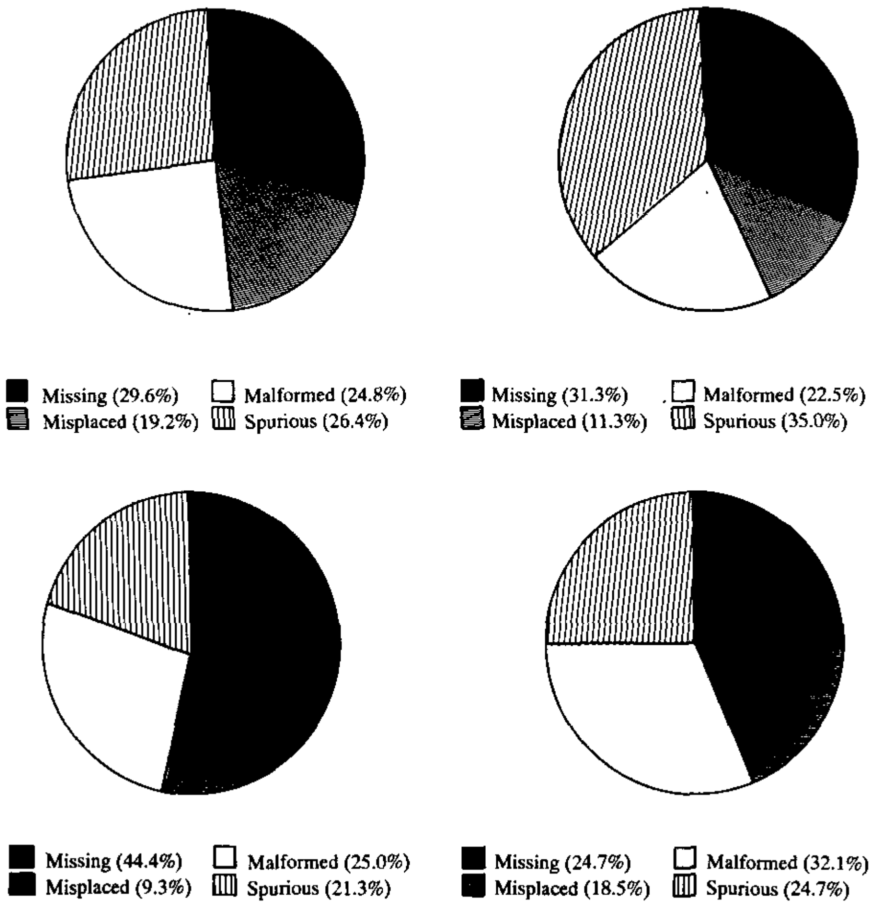

FIGURE 6. Plan difference errors.

7. Correlation of plan composition and language constructs errors

A Plan is the conceptual representation of a program and its subdivided steps in solving a problem. A program usually requires several Plans to reach the solution and these Plans must be put together properly. Novice programmers develop some of these conceptual model representations (Plans) and their compositions as they learn programming. Language Constructs are used as building blocks to form a Plan. The errors from the Language Constructs (Table 1) and Plan Composition experiments (Table 4) demonstrate a high linear correlation among these types of errors for all the programming languages in question. A strong positive correlation between Language Constructs and Plan Composition errors ranging from 0.91 to 0.97 across programming languages indicates that programmers who make more Plan Composition errors also tend ot make more Language Constructs errors and vice versa. For example, in Pascal, subject ID# 17 had one error in Language Constructs, and no errors in Plan Compostion. Subject ID# 9 had nine errors in Language Constructs and 10 in Plan Compostion. The results are very similar for all the other languages.

In the following regression analysis, 1.3, 1.1, 1.6 and 0.9 are the slopes of the regression lines; 0.0, 0.3, −1.0 and 0.5, are the intercepts where $y = mx + b$ or Plan = (slope $* x$) + intercept:

plan = 1.3 * construct + 0.0	$r = 0.96$ (Pascal)
plan = 1.1 * construct + 0.3	$r = 0.91$ (C)
plan = 1.6 * construct − 1.0	$r = 0.97$ (FORTRAN)
plan = 0.9 * construct + 0.0	$r = 0.92$ (LISP)

The correlation graph for Language Constructs and Plan Composition errors for each of the languages is shown in Figure 7. Each dot represents the intersection of Language Constructs and Plan Composition errors. The larger, open dots represent two or more data points in the same intersection. The existing straight line of correlation can be attributed to the fact that novices use Language Constructs to build the Plans. Similarly, the Plans are viewed and expressed in terms of Language Constructs.

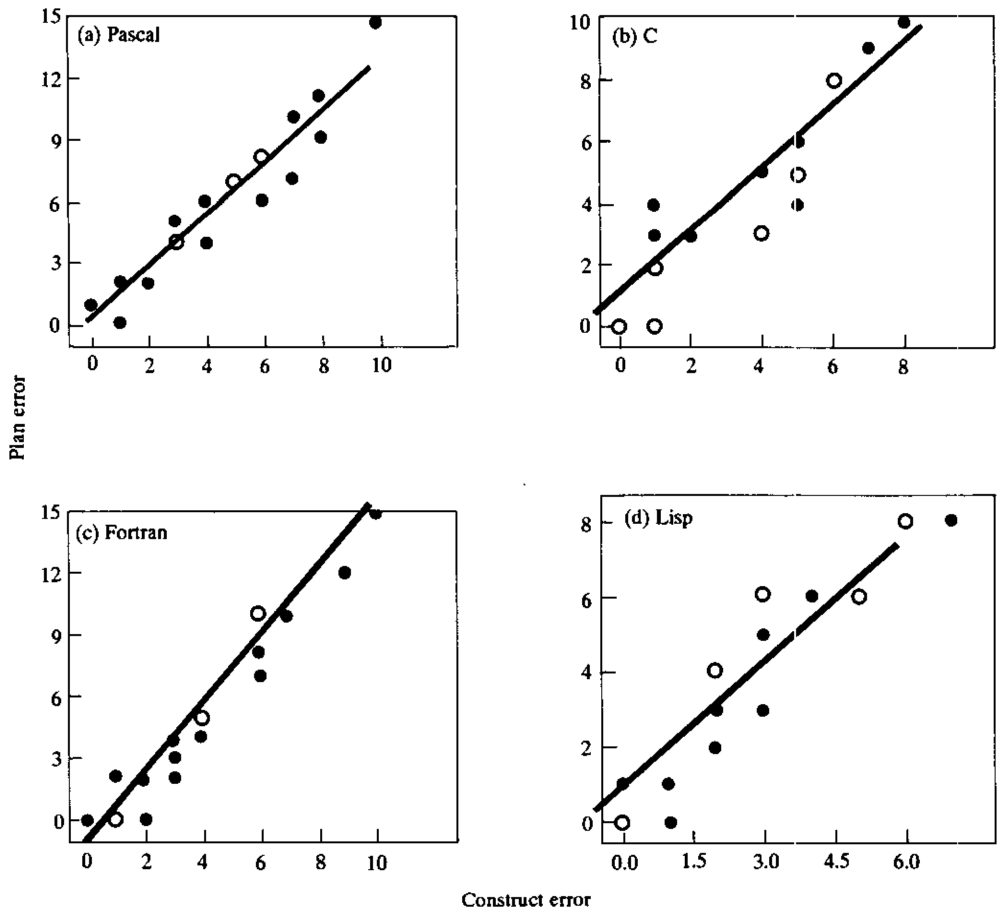

FIGURE 7. Correlation of Plan composition errors versus Language Constructs. (a) Pascal, (b) C, (c) FORTRAN, and (d) LISP. Open circles represent two or more data points at the same intersection.

Several prior studies suggest that most of the programming errors occur in Plan Composition rather than in Language Constructs. It should be noted however, that when a problem is given as an experiment for both Language Constructs and Plan Composition, such as the average "rainfall" program, students often have the option of using those Language Constructs with which they are most familiar and because this inevitability suppresses the number of errors in Language Constructs, it impedes the process of genuine error evaluation. A more reliable rate at which Language Constructs errors occur is determined by designing experiments that are related directly to the comprehension of Language Constructs. Some of the current research shows that students have more problems understanding simple loops than understanding simple problems (Shackelford & Badre, 1993). This supports the conviction that the knowledge of Plan Composition and Language Constructs is indispensable to the understanding of basic programming.

8. Conclusion

This study was conducted with 80 novice programmers divided into four equal groups. Each group used one of the four following languages: Pascal, C, FORTRAN and LISP. The subjects were taking introductory courses in these indicated languages at the time. Each group participated in two experiments, one known as Language Constructs and the other as Plan Composition.

In the Language Constructs experiment, several small programs, called segments, were given to the students to run mentally and record the output with the given input. These test were chosen in such a way as to examine the basic understanding of Language Constructs, such as input and output, decision making, loop and specific features of the languages.

In the Plan Composition experiment, a common problem known as "rainfall" was posed. The "rainfall" problem is Plan-oriented as is typically used in introductory programming courses where the average, minimum and maximum amount of rainfall is determined. Prior studies have used the "rainfall" problem similarly. A Plan is a problem-solving paradigm which consists of one or more related steps performing a task. A Plan may be comprised of many subsidiary Plans, which themselves become Plans. The study of the Plan Composition experiment investigates the cause of errors based on how Plans are put together. Plan errors may occur as a result of steps that are missing, malformed, misplaced or spurious. Language Constructs serve as building blocks for Plans such as IF (decision making), WHILE (repetition), and READ and WRITE (input and output). A Plan can be implemented in more than one way using different Language Constructs within the framework of the same language. In addition, because of peculiar features of a language, a Plan may be implemented uniquely, as in LISP where recursion is more commonly used. Moreover, the same plan may be developed in different programming languages using similar Language Constructs.

The errors for Language Constructs can be categorized as follows: in Pascal, Logical IF, Input, and REPEAT...UNTIL loop; IF with "=" Operator, Logical IF and DO...WHILE loop; in FORTRAN, IF without Else, Nested IF and Logical IF; in LISP, Logical IF, Compound IF and DO Loop.

The analysis of Language Constructs errors indicates that the majority of these errors can be generalized in the following three categories: IF statements, Loops, and specific language features. The IF statements with logical operators were the most troublesome in all languages. This is primarily due to the students' misunderstanding of the difference between logical AND and OR. In natural languages, these notations are variously used, whereas in computer languages the difference between is formally precise.

Loop errors mainly occurred as a result of the misinterpretation of loop termination. The termination of a loop is based on the true or false value of the condition. For example, in Pascal the REPEAT...UNTIL loop terminates when the condition is true; on the other hand, the WHILE loop terminates when the condition is false.

Some of the errors occurred as a result of the specific language features, for example, Input in Pascal, use of "=" operator in C, and DO loop in LISP. Input in Pascal is format free and a blank space between two data, where the last is a character, causes a problem. The blank space would be read as the value of the character. The equality operator in C uses two equal signs, "= =". One equal sign, however, indicates assignment and is permissibly embedded in the IF statement. The steps involved in the DO loop in LISP are written linearly, but the execution action will not follow this order.

In the analysis of Plan Composition errors, the most frequent Plan error for all languages was the Missing Guard IF. The other Plan errors can be itemized as follows: in Pascal, Misplaced Update and Malformed Loop; in C, Spurious Output and Malformed Initialization; in FORTRAN, Missing Initialization and Misplaced Loop; in LISP, Misplaced Update and Spurious Output.

Beginner programmers just run the programs for a given data set, but they fail to perform error checking. The Missing Guard IF Plan error occurs as a result of the student's negligence in checking the invalid data. For the novice programmer, simply solving a problem is difficult enough.

The Misplace Update Plan error occurred as a result of the improper location of the assignment expression, such as the placement of the average computation inside the loop. Initialization errors occurred because of the students' assumption that the language would automatically set the proper variable initialization. The Spurious Output errors result from students displaying unrequested information. In LISP, however, the return value of the function will be displayed automatically.

The comparison between the number errors of the Language Construct and Plan Composition experiments for each student exhibits a high linear correlation among those specific types of errors for the programming languages discussed. Students who had more errors in the Language Constructs experiment also made more errors in the Plan Composition experiment. In the same manner, students who had minimum errors in Language Constructs also had a minimum number of errors in the Plan Composition experiment. The understanding of Language Constructs contributes indipensably to the building of Plans. Similarly, novice programmers expressed the Plans in terms of the Language Constructs.

This study indicates that novice programmers have difficulties in both Language Constructs and Plan Composition. It is incumbent on instructors to pinpoint these potential pitfalls and help novice programmers to be aware of and to overcome these difficulties. For example, error checking must be emphasized and more stress placed upon the specific features of the languages where misconceptions exist. The study concludes that the Plan Composition capability and semantic interpretation of the Language Constructs are mutually correlated to a significantly high degree. Therefore, the teaching and presentation of Language Constructs and Plan Composition must be accomplished concurrently.

References

Anderson, J. & Jeffries, R. (1985). Novice Lisp errors: undetected losses of information from working memory. *Human–Computer Interaction,* **1**(2), 107–131.

Bayman, P. & Mayer, R. (1983). A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM,* **26**(9), 677–679.

Bonar, J. (1985). *Understanding the bugs of novice programmers.* Ph.D. Dissertation, University of Massachusetts, MA, U.S.A.

Bonar, J. & Cunningham, R. (1988). Bridge: tutoring the programming process. In J. Psotka, L. Dan Massey & S. Muter, Eds. *Intelligent Tutoring Systems: Lessons Learned.* Hillsdale, NJ: Lawrence Erlbaum.

Detienne, F. & Soloway, E. (1990). An empirically-derived control structure for the process of program understanding. *International Journal of Man–Machine Studies,* **33,** 323–342.

Ebrahimi, A. (1989). *Empirical study of errors by novice programmers and design of Visual Plan Construct Language (VPCL).* Ph.D. Dissertation, Polytechnic University, NY.

Ebrahimi, A. (1992). VPCL: a visual language for teaching and learning programming. (A picture is worth a thousand words). *Journal of Visual Languages And Computing,* **3,** 299–317.

Eisenberg, M. (1987). Understanding procedures as objects. In G. M. Olson & E. Soloway, Eds. *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Ericsson, K. & Simon, H. (1983). *Protocol Analysis: Verbal Reports as Data.* Boston, MA: The MIT Press.

Frye, D., Littman, D. & Soloway, E. (1988). The next wave of problems in ITS. In J. Psotka, D. Massey & S. Mutter, Eds. *Intelligent Tutoring Systems: Lessons Learned.* Norwood, NJ: Lawrence Erlbaum.

Johnson, W. (1986). *Intention-Based Diagnosis of Novice Programming Errors.* Los Altos, CA: Morgan Kaufman.

Johnson, W. & Soloway, E. (1983). *BUG CATALOGUE: I.* Dept. of Computer Science: Cognition and Programming Project, Yale University.

Klerer, M. (1984). Experimental study of a two-dimensional language vs. Fortran for first-course programmers. *International Journal of Man–Machine Studies,* **20,** 445–467.

Knuth, D. (1971). An empirical study of FORTRAN programs. *Software-Practice and Experience,* **1,** 105–133.

Ostrand, T. & Weyuker, E. (1984). Collecting and categorizing software error data in an industrial environment. *The Journal of Science and Software* **4,** 289–300.

Shackelford, R. L. & Badre, A. N. (1993). Why can't smart students solve simple programming problems? *International Journal of Man–Machine Studies,* **28,** 985–997.

Schank, R. & Abelson, R. (1987). *Scripts, Plans, Goals and Understanding.* Hilldale, NJ: Lawrence Erlbaum.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM,* **29**(9), 850–858.

Spohrer, J. & Soloway, E. (1986*a*). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM,* **29**(9), 624–632.

Spohrer, J. & Soloway, E. (1986*b*). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Lyengar, Eds. *Empirical Studies of Programmers. Proceedings of the First Workshop.* Norwood, NJ: Ablex.

SPOHRER, J., SOLOWAY, E. & POPE, E. (1985). A goal/plan analysis of buggy pascal programs. *Human–Computer Interaction,* **1**(2), 163–207.

WEINBERG, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

WEISSMAN, L. (1974). *Methodology for studying the psychological complexity of computer programs*. Ph.D. dissertation, University of Toronto, Canada.

YU, C. & ROBERTSON, S. (1988). Plan-based representation of Pascal and Fortran code. Edited by E. SOLOWAY, D. FRYE & S. SHEPPARD, Eds. *CHI 88 Conference Proceedings*. Addison Wesley.