The creation of plan schemas is examined in a naturalistic, longitudinal study of problem solving. Ten novice programmers each wrote eight Pascal programs to solve a series of problems. Their protocols were analyzed to determine how declarative programming knowledge was used to create simple procedural program plans, and how the simple plans were used to create complex plans. Plan creation showed a process of backward development, from the goal to the plan focus, that part of the plan that directly implements the goal. Once defined, it is expanded to create the complete plan, showing backward development of plan parts from the focus. Once the plan is complete, it may be stored as a plan schema and retrieved in subsequent problem solving. The plan will then show forward development as it is implemented in schema order, the order in which plan pieces occur in the finished program. The change from backward development during creation to forward development after retrieval was strongly evident in the statistical analysis of the protocol data. Previous studies of novice programming, which showed only forward development, are explained as special cases of this more general model, cases in which schema knowledge was available to the problem solver.

The dominant view of both novice and expert programming emphasizes forward design of a solution. Such design is possible when a schema can be retrieved and implemented to provide a solution to a problem. Given a goal to achieve, the expert programmer can simply retrieve the appropriate plan from memory, or the novice can identify the required plan from a textbook. The plan can then be implemented in the current problem, and pieces of the plan will appear in the order they occur in the retrieved schema.

When a template cannot be applied, forward design breaks down and a new solution must be created. Given a goal to be achieved, the process of plan creation begins with a skeletal plan that directly implements the goal.

This research was funded by a grant from Yale University. I wish to thank David Littman and Jim Spohrer for their many comments and criticisms as this work evolved. I also wish to thank Robert Sternberg, Roger Schank, and Elliot Soloway for their assistance and support at Yale. Any shortcomings in the final version of the paper are, of course, my responsibility.

Correspondence and requests for reprints should be sent to Robert Rist, Department of Computer Science, University of Technology, P.O. Box 123 Broadway, Sydney, New South Wales, Australia, 2007.
and provides a focus for further problem solving. The focus is usually not a complete solution, so it must be extended to define the full plan. The plan can then be stored in memory as a single chunk of programming knowledge, a plan schema, and retrieved and applied in later problem solving. This article explains the behavior seen in program design as a result of the intricate interplay between plan creation and retrieval.

The development of a program may be traced by defining the plans used in the program and noting the order in which pieces of the plans appear during the process of design. The development of expertise may be traced by comparing the process of plan creation to later retrieval and use of a plan in new problems. Plan creation is based upon expansion of the plan focus, and creates an order of emergence of plan pieces corresponding to backward and bottom-up program design. Plan retrieval and implementation is based upon expansion of the plan schema, and pieces of the plan appear in schema order, creating a pattern of forward and top-down development of the program. With increasing experience, planning changes from plan creation to plan retrieval, and program design changes from backward and bottom-up, to forward and top-down design.

The article describes the process of program design in terms of the knowledge used by the programmer. It does so in three steps. First, the literature on program design and representation is reviewed. Then the knowledge used in program design is defined, and the design process is illustrated with a series of protocols. A formal model of design, focal expansion, is then presented and tested by statistical analysis of 80 protocols from an experimental study. Finally, some of the implications of the research for programming, and for general models of planning, are discussed.

MODELS OF PROGRAM DESIGN

Most researchers in the field of program design report their results in terms of the model of top-down design. In top-down design, the program is conceptualized as a tree structure, which is constructed from top to bottom. At the top of the tree is an abstract idea about the solution, at the bottom of the tree is the program code. Top-down design begins with the abstract solution, which is progressively refined and made more concrete. At each step in the development process, one part of the tree is selected and expanded into a more detailed form, and the expansion process continues until the solution can be implemented in program code. The structure of the tree is preserved at all levels of detail, from abstract to concrete.

Several studies are described in this section that demonstrate top-down design in novice and expert programmers. More recent studies have shown that program design does not always show this sequence of development. There are times when experts abruptly change their level of detail during design, and times when novices do not show a coherent design approach. It
is proposed that these apparently contradictory studies can be explained by defining the knowledge used in program design, and showing how differences in knowledge create the different design strategies.

Top-down design was first studied by Jeffries, Turner, Polson, and Atwood (1981). They concluded that both novice and expert programmers designed the program from front to back, using a strategy that decomposed the problem into subproblems. Novices did an initial decomposition to sketch out the solution and coded the first part of this solution without further analysis. They then coded the next part of the solution, then the next and so on; an abstract view of the total solution was never apparent in their protocols. In terms of the tree formalism, the same part of the tree was expanded at progressive levels of detail, until that part of the solution could be implemented in program code. This behavior can be described as depth-first search of the tree; novices developed the program using top-down, forward and depth-first search. Experts also used a decomposition strategy and created the solution by top-down design, but they maintained the same level of detail for all parts of the solution as the tree was developed. At any stage in design, all the parts of the design were equally abstract, or equally concrete. They generated the solution by top-down, forward and breadth-first search.

The use of top-down design by experts was supported and extended by Adelson and Soloway (1985), who compared expert programmers designing in familiar and unfamiliar domains. They reported a sequence of stages in expert solution development, as an initially sketchy solution was expanded in a systematic manner through a series of more and more detailed models of the solution. When working in a familiar domain, the designers simulated the solution at each level of expansion, to ensure that all the required pieces were present at that particular level of description. This process of systematic expansion created a pattern of top-down, forward, and breadth-first development. In new domains, the designers lacked a complete model of a solution, and were forced to revert to simpler, local models. Within each local model, designers simulated the partial solution to see if it worked, then combined the pieces to form a full solution.

The use of top-down design by novices was supported and extended by Anderson and others (Anderson, Farrell, & Sauers, 1984; Pirolli, 1986; Pirolli & Anderson, 1985), who described both the process and the knowledge used in program design. They concluded that design operated on a program template, from the text or the programmer's memory, that was retrieved and implemented to provide a solution to the problem. These templates were defined as abstract program structures, such as the structure of a LISP operator, a general function or a recursive function. The template was identified and each slot in the template expanded in full, before the next slot was attempted. The order of development of solution pieces created by such a process produced a pattern of top-down, forward and depth-first design.
The difference in the search processes between novices and experts may be explained as a result of the knowledge that individuals bring to bear on the process of design. Novices have only low-level representations of programming knowledge, whereas experts have representations at both abstract and concrete levels (Adelson, 1981, 1984; Schneiderman & Mayer, 1979). The ability to represent parts of a solution underlies the ability to design a complete solution using those parts, so the lack of knowledge of the novices precludes the use of abstract design. The conclusion drawn from these studies is that programs are designed by top-down and forward expansion. Within this framework, experts show breadth-first design, but novices can design only at the detailed levels, and so show depth-first design.

Recent research has shown that the model of top-down and forward search is not always applicable. There are times when an expert shifts from abstract to detailed design and back again, or shows little evidence of top-down design. Visser (1987) analyzed the behavior of a professional designer who used a program, previously employed for a similar task, to guide the design of a new solution. The designer adopted a modular, top-down strategy at the level of the overall program structure, but a bottom-up strategy to create the new details of the program code. Guindon, Krasner, and Curtis (1987) reported a study of expert designers in which top-down design was rarely seen. The most expert programmer studied did show a process of successive refinement for a problem that was familiar to him, but performed much more exploratory design for unfamiliar problems. The other programmers studied showed a process of serendipitous design, characterized by the creation of solution pieces at different levels of detail, without having previously decomposed the problem into subproblems. Each of these pieces represents a part of the total solution, but the connection between the pieces is initially unknown. To construct a solution, the disparate pieces must be integrated into a single framework. The lack of knowledge created behavior similar to that observed by Adelson and Soloway (1985) when expert designers were faced with new and difficult problems.

On theoretical grounds, it seems that novice programmers, when they are just starting to learn to program, cannot employ stepwise refinement as a design tool. They have not yet developed the program templates that allow a problem to be matched with a solution and decomposed into a set of connected pieces (Soloway, 1986). When writing a program, they tend to flounder and search for a solution with little overall plan or organization.

The results of the studies are summarized in Table 1, categorized by the analysis technique used by the experimenters, and whether the subjects were novices or experts in the domain. When knowledge was available to guide design, behavior showed forward and top-down design. When knowledge was not available, the results are more complex. At the level of program code, examples of design were seen that did not correspond to top-down design. At the more abstract levels described by the verbal protocols, Adel-
SCHEMA CREATION

TABLE 1
Support (+) or not (−) for Top-down Design:
Categorization of Studies by Dependent and Independent Variables

<table>
<thead>
<tr>
<th>Knowledge available</th>
<th>Verbal Analysis</th>
<th>Code Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Jeffries et al. (1981)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Adelson &amp; Soloway (1985)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Guindon et al. (1987)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Viser (1987)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knowledge not available</td>
<td>+Adelson &amp; Soloway (1985)</td>
<td>−Soloway (1985)</td>
</tr>
<tr>
<td>−Guindon et al. (1987)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

son and Soloway (1985) observed top-down, forward, and local design, Guindon et al. (1987) observed serendipitous design, and Visser (1987) observed bottom-up design. What can explain the different results found by these investigators?

Analysis of verbal protocols for complex problems is necessarily imprecise. First, the verbal discussion is usually not explicitly related to the final program code; Adelson and Soloway may have observed the same behavior as the other researchers, but described it differently. Second, the number of levels of abstraction within the solution is undefined; Adelson and Soloway may have described planning at a different level of abstraction than the other researchers. Unless the tree structure can be precisely specified, it is difficult to compare behavior within a subject at different levels of abstraction, between subjects in a single study, and across studies.

It is proposed that both novice and expert programmers possess abstract knowledge of program structure, and apply this knowledge to design an abstract solution using forward and top-down design. Novices adopt a very local strategy, which places a minimal load on their memory, and design the same part of the solution at different levels of abstraction, creating a pattern of depth-first solution design. Experts use breadth-first design to map out and develop a complete solution to the problem. Abstract knowledge is not useful for the detailed design of program code, however, so top-down design may break down. At that point, specific program plans have to be retrieved from memory, or created. If the plans are retrieved and implemented, forward and top-down design continues to the level of program code. If the plans are created, development of solution pieces will show a pattern of local, backward, and bottom-up design.

To test this proposal, abstract and detailed plan knowledge must be precisely defined, and plan expression must change with expertise, from backward and bottom-up to forward and top-down. The cognitive structure of programs is defined in the next section at four levels of detail: at the level of
a line of code, a simple plan, a complex plan, and the program. Plan design is then illustrated by a series of verbal and code protocols, and the change in plan usage with expertise is described and tested.

**THE COGNITIVE STRUCTURE OF PROGRAMS**

Descriptions of program structure and design have often been driven from the final artifact, the finished program. This approach biases the researcher in two ways. First, there is the temptation to look for global and abstract structure, in an attempt to capture the large or important divisions within a program. Second, there is the temptation to report results in terms of deviations from an expert model, and thus to describe novice behavior in terms of what novices do not do. The developmental approach followed here has a different set of biases. Because the focus of attention is on individual design decisions, the level of reporting describes the way in which a novice builds up a program from small and disparate pieces. Only at the end of the construction process is the final and abstract form of the solution apparent.

The structure of a program must be described at several levels to show how it is created from simple pieces of knowledge. At the finest level of detail, individual pieces of knowledge are combined to create a single line of code. At a less detailed level, individual lines of code are combined to form simple plans. At a more global level, simple and complex plans are merged to achieve the goal of the problem. At the most global level of description, a program has an abstract structure, corresponding to a program template. As small pieces of knowledge are combined into lines of code, lines into simple plans and simple plans into complex plans and programs, the complexity of the artifact obviously increases. The cognitive complexity is kept manageable by the use of chunking; the chunks of knowledge used at each level of description hide the complexity of the level below it. The novice has to create knowledge at all four levels in order to build a program, but this knowledge can be retrieved instead of created as expertise develops.

**From Code Fragment to Single Line**

The creation of just a single line of a code requires a great deal of reasoning and planning. Mayer (1987) described the conceptual structure underlying a line of BASIC code and found that the smallest piece of knowledge used in program understanding is a transaction. A transaction can be described using three categories: what operation takes place (action), where it takes place (location) and what object is acted upon (object). As an example, 10 transactions are required to understand the line of code, LET B = A + 1, which simply adds one to an integer (A) and stores the resulting sum in memory (in B). First, the integer and the increment must be defined and stored in a temporary memory (4 steps), and added together (1 step). Then
the location of the sum must be defined, the sum placed in that location and deleted from the temporary memory (3 steps). Finally, control must be transferred to the next statement and that statement executed (2 steps).

Within a single line of code, it is proposed that there is a central or focal code fragment that represents the most important action in the line. This focus is the part of the code that immediately implements the current goal. For the example above, the programmer's goal is to increment a number, so the focus is the code fragment that does the incrementing, "+ 1." This fragment is far from a complete plan, but it provides a focus around which the rest of the plan can be developed.

From Line of Code to Programming Plan
At the next level of cognitive complexity, single lines of code are combined to create a single program plan. Just as a code fragment needs to be extended to generate the full line, a line needs to be extended to generate the full plan. To extend the previous example, one of the most common plans in programming is to count up the number of times that some event occurs. The goal of the plan is to create a total count. The focus of the plan consists of adding one to a running total every time the event occurs (+ 1) inside a program loop. This focus must be extended to a full line of code, that adds one to the count so far (count + 1) and stores the total (count: = count + 1). The total must be set to zero (count: = 0) before the loop, for the count to produce the correct result.

Five common plans, which make up much of the content of any program, are shown in Table 2. These are the prompt, label, running total, found, and loop plans. The prompt plan achieves the goal of reading in a single value. The focus of the plan, the code that reads in the value, is the READ statement; to tell the user what value to type in, it needs to be extended by a prompt. The label plan is similar: the focus writes out a single value. To tell the user what the value indicates, a label statement needs to be added. The count plan is one example of the running total plan; the plan can also be implemented as a sum plan, if a variable value is added to the running total.
The found plan tests a series of values inside a loop, looking for a certain condition (say a value greater than 100). If the test ever succeeds, a flag is set to true (the focus), and the value of the flag tested after the loop. For the plan to work correctly, the flag has to be set to false before the loop. The loop plan is illustrated by a read loop, in which a read statement does the work (reads in a single value) and is generalized by the loop to read in many values.

The table has two sets of headings, the first reflecting plan creation, the second plan retrieval. In plan creation, the plan is developed backward from the goal, to the focus, to the extension of the plan. In plan retrieval, the pieces of the plan reflect the final form of the plan schema, first the Input or Initialization (I), then the Calculation (C), and then the Output (O).

The use of these plans in program design defines two dimensions. For the first four plans, the shift from focus to schema order corresponds to a change from backward (CI) to forward (IC) plan expression. For the loop plan, this dimension is not appropriate because the contents of the loop both follow and are controlled by the loop statement. The content of the loop deals with a single value, and the loop statement generalizes this action. For the loop plan, the change from focus to schema order corresponds to a change from bottom-up (CI) to top-down (IC) plan expression.

From Program Plan to Program
At the next level of description, basic plans make up complex plans, and simple programs. The way in which basic plans are connected to create a program has been investigated by Soloway and colleagues (Soloway, 1986; Soloway, Bonar, & Ehrlich, 1983). They define programming knowledge in terms of plan schemas such as those in Table 2—program plans that implement a common problem goal—although more complex plans are also discussed (Soloway, 1985). For Soloway, the program has a complex goal and plan structure, in which the plan schemas provide "canned" solutions for common goals. These plans can be used to achieve the goals of the problem and, when merged together correctly, create the structure of the program. No model of design using these plans has been proposed by these researchers, because their focus of interest has been on knowledge representation and on program errors that misuse the knowledge.

Abstract Program Structure
The visible structure of a complete program is created by the complex plans that make up this final form. Knowledge about this structure, in the form of program and function templates, was used by Anderson et al. (1984) to explain the process of program design. Templates are powerful design tools, but their internal structure is hidden. The decisions and dependencies that created the template originally are compiled out of the final representation (Anderson, 1982). To be useful in design, the underlying deep structure
must be understood as well as the abstract or surface structure. These two types of knowledge are very different. Kessler & Anderson (1986) reported a study in which all subjects learned a program template for recursion, but several of them never derived the implicit information about flow of control within the template. These subjects attempted to solve a problem by mapping from the surface structure of the template to a solution, without understanding how the program worked. Successful subjects were able to derive the implicit structure of the template from the surface structure. Knowing both types of information, they were able to map the template onto new problems correctly.

Abstract program structure has been analyzed in language similar to that of role structure (ICO), using the terms "input goal," "calculate goal," and "output goal" (Spohrer, Pope et al., 1985; Spohrer, Soloway, & Pope, 1985). The terms goal and plan are used by these researchers to refer to the abstract structures, as well as the specific plans that make up the detailed structure of the program. A strong distinction will be made in this article between the surface structure of a program, and its deep structure. The surface structure consists of a template or schema for the program; here, it consists of the three abstract roles that make up a plan or a program: Input, Calculate and Output (Rist, 1986a). The deep structure of a program consists of the specific plans, which when merged together in the correct way, achieve the goal of the problem and make up the surface structure of the program. The two descriptions of the program create very different solution strategies. The surface structure is used to guide top-down design, and the deep structure creates the program by bottom-up design.

The Development of Programming Knowledge

The model of solution design proposed here describes plan generation at four levels, from code fragment to final program. Given a goal, the novice retrieves a fragment of code that immediately implements the goal, but is far from a complete plan. This piece does provide a focus for further problem solving, however, and allows the other pieces of the plan to be added in support of the focus. Programming plans are created outward from the focus, to a focal line, to the complete plan. The plans are then merged to define the full program.

Expertise is created by this process of plan generation. The novice programmer begins with no knowledge of plans, and struggles to define a single line of code. Once a line has been defined, the focus of design shifts to the design of a single plan. Once a plan has been defined, the programmer must integrate the plan into the current complex plan or program. Once a complex plan has been completed, the next complex plan has to be added to the program. At each level, the complexity of the program increases. At a given point in time, a programmer will have stored the knowledge required for design at a single level, and be able to retrieve this knowledge to create a
solution at the next level. As expertise develops, detailed planning at one level disappears and is replaced by retrieval and planning at the next level of design. Abstract knowledge, however, is available as soon as a single plan has been created. Experts possess both abstract and specific plan knowledge, and apply this knowledge to create the program by forward and top-down design. Novices also possess this abstract knowledge, but their behavior is a product of the interaction between abstract role structure and specific plan structure.

The creation of specific program plans is illustrated in several protocols below, at three levels of detail: where fragments of code make up a single line of code, lines of code make up a basic plan schema, and basic plans make up a complex plan. To illustrate the use of role structure in program design would require a long and tedious example, so it is omitted from the protocols.

MOTIVATING THE MODEL: SAMPLE PROTOCOLS

Two protocols are used in this section to illustrate the process of plan development. The first protocol shows the development of a basic plan, the sum plan, from the plan focus to the focal line and from the focal line to the full plan. The second protocol shows the development of a complex plan, in which each of the basic plans is retrieved, but the complex plan is created by focal expansion. Taken together, the examples illustrate three levels of plan extension, from the plan focus, from a focal line, and from a focal plan. The protocols are drawn from a set of 10 novices in their first programming course; details of the subjects and procedure are given in a later section.

Creation of a Basic Plan

The first protocol describes the development of the sum plan. The goal of the plan is to find the sum of a set of numbers. The focus of the plan, the part that immediately implements the goal, is the statement that actually adds up the numbers and calculates the sum. The plan was first used in the fourth problem of the study, referred to here as the Noah problem. The problem asked subjects to find the average rainfall and the number of rainy days, given the rainfall each day for a month.

The creation of the sum plan is illustrated in Table 3. The first two columns of the table show the behavior of the subject; the verbal protocol is shown first, the corresponding program code is shown next. The last two columns at the right of the table relate the behavior to the model of plan design described above; the first column uses the process terms of the model, the second column shows the role structure of the plan, taken from Table 2.

The subject stated the goal, to calculate the total rainfall, and coded the focus (+ rain) of the plan. The focus was then extended to the focal line in three steps. First, a variable was defined to contain the total, the final value
Protocol Showing Plan Development from the Plan Focus, with Backward Plan Expression

<table>
<thead>
<tr>
<th>Verbal Statements</th>
<th>Program Code</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N7) &quot;For rain itself, rain is going to equal... I need... I need something... I need something there that keeps... I'm setting sum equal to the amount of rain from before, OK... sum will be a problem, I'll come back to it later... [other goals solved]... OK, do sum gets sum plus rain, or even better... and now set that to zero&quot;</td>
<td>rainfall:=... rainsum:=rainsum+rain; rainsum:=0;</td>
<td>goal O, focus C, extension C, focal line C, extension 1</td>
</tr>
</tbody>
</table>

used in calculating the average; note that at this point, the plan to calculate the total has not yet been defined. The code to add a single value of rain to the sum so far was then added, defining a full line of code. All the pieces of the focal line were now defined, but they were not correctly connected to each other. The sum so far and the total value, conceptually separate, had to be merged together into the same variable, here called “rainsum.” This merge created the final form of the focal line for the plan, and the full plan was then defined by initializing the sum to zero before any values were added to it.

Creation of a Complex Plan

The second protocol illustrates how a complex plan is built from basic plans. The plan has two parts, a loop statement and the contents of the loop. The contents of the loop may be considered as a set of actions to solve the simple goal of reading in a single value, which are generalized by the loop. Informally, such a strategy is to do it for one case, then extend the plan to the general case; such a strategy creates a pattern of bottom-up development of the plan pieces.

A simple read loop, that reads in 30 values of rainfall, was shown in Table 2. The goal of the plan is to read in a set of numbers. The focus of the plan, the part that immediately implements the goal, is the statement that reads in a single number, the READ statement. This statement is actually the focus of two plans, the prompt plan and the read loop plan. Assuming that the prompt plan is simply retrieved and implemented, it needs to be extended to read in a set of numbers by placing it within a program loop. The plan was first used in the Noah problem, in the form shown in Table 2. The second use of the read loop, illustrated in Table 4, extended the simple plan to read in an unknown number of values.
TABLE 4  
Protocol Showing the Creation of a Complex Plan from Retrieved Basic Plans:  
All Basic Plans Show Top-down or Forward Plan Expression

<table>
<thead>
<tr>
<th>Verbal Statements</th>
<th>Program Code</th>
<th>Interpretation</th>
</tr>
</thead>
</table>
| *(N2)* "The first thing I want to do is to get the original word... You might want to use an array for the letters in the word, so you have..." | TYPE letters=array[1..20] of char;  
VAR word: letters;                                                                 | goal           |
| Now get the original word, so say WHILE... use a REPEAT loop... give a prompt first... then REPEAT... | write ('Please enter a word');  
repeat                                                                         |               |
| umm... you want to read it in... you want a counter... initialize a counter, and then you want to repeat... so you read it in, and... | read (...); repeat (word[i]);  
i:=0; [before repeat]; i:=... [before repeat]; i:=1; [before repeat]; i:=i+1; [before repeat]; until (word[i]=' '); |               |
| ... and then increment it by 1... "uh-uh, until..."                               | use O_count; use O_read;                                                      |               |

For those readers who are unfamiliar with Pascal, a short explanation may be useful. The subject's plan is to read in letters, one at a time, and store them in an array until the end of the word is reached. The end of the word is indicated by the user typing in a blank character. An array is a set of storage locations in the computer that can be addressed by an index; assuming that the name of the array is "word," the first location in the array can be accessed by the address word[1], the second by word[2], and so on. All the locations in the array can be accessed, one at a time, by using the address word[i], where "i" takes on the values 1, 2, 3, and so on; "i" is called the index of the array.

The individual plans within the complex plan all showed schema expansion. Development of the loop plan showed top-down expression: first the loop statement (repeat) was coded, then the focus of the plan, the read statement. The prompt plan showed forward expression, with the prompt statement preceding the loop and the content of the prompt referring to the goal to enter the word, not to read in a single character. The count plan also showed forward expression, as the count was first initialized to zero, then the increment coded.
Development of the complex plan showed a series of cycles of plan retrieval, implementation, and extension. The goal of the plan was to read in a series of letters. The focus of the complex plan was the read statement, which retrieved the read loop plan. This plan was implemented, but in the middle of coding the read statement, it was seen to be inadequate. It had to be extended by the addition of an index for the array, so the count plan was retrieved and implemented. In the middle of coding the calculation, the plan was seen to be incorrect; the first letter would have been stored in word[0], which does not exist. The count was then initialized to one, not zero, as defined by the normal count plan schema.

The final plan is still not quite correct. As written, the code will read in the blank character and store it, say as the 10th letter, increment the index to 11, and test the 11th location of the array to see if it is a blank. The subject noted a bug in the plan to store the letters, and changed the count plan to overcome this problem. In doing so, a bug was introduced into the plan to stop the loop. This type of error is common in novice programmers, and is called the "off-by-one" bug (Spohrer, Pope et al., 1985). One more extension of the plan is needed to correct the error: the value of word[i−1], not word[i], could be tested at the end of the loop, or the increment moved to a position before the read statement. The subject did not notice the new bug because the original problem was solved, and did not explore the implications of the solution.

In this example, both planning and debugging behavior were local to the context of the current problem. The three basic plans showed forward development, but the complex plan as a whole showed development from the focus. The behavior of the subject could be described as depth-first search, because each plan was developed in isolation, but it was not top-down. The subject did not expand an abstract schema, because each new step in design was a surprise, and the subject did not have a schema for the complex plan. An explanation for the subject’s behavior lies in the detailed interaction between the retrieval of specific plans, and the process of focal expansion.

Abstracting from the Protocols
Three levels of plan creation were illustrated in the protocols. At each level, the plan focus began the creation process, and was extended in the remainder of the protocol. In the first protocol, a single line of code was built up from an initial plan focus, and a basic plan was built up from the focal line. In the second protocol, a complex plan was built up from basic plans. The pieces used to build a plan were different, but the process of focal expression remained constant.

Two dimensions of plan expression were demonstrated in the protocols, bottom-up versus top-down, and forward versus backward. The forms and levels of plan expression are summarized in Table 5. Bottom-up develop-
TABLE 5
Description of Plan Expression in the Protocols, within a Line of Code and within a Plan

<table>
<thead>
<tr>
<th></th>
<th>Bottom-up</th>
<th>Backward</th>
<th>Top-down</th>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within a line</td>
<td>N7, sum</td>
<td>N7, sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Within a plan</td>
<td>N2, loop</td>
<td></td>
<td></td>
<td>N2, prompt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N2, count</td>
</tr>
</tbody>
</table>

ment of the focal line of the sum plan was shown in the first protocol, for subject N7, as well as backward plan development from the focal line. The second protocol (N2) showed top-down plan expression, as the loop plan was retrieved and coded, then the contents of the loop were coded. It also showed forward expression for the prompt and count plans, as the plans were retrieved and implemented with little discussion about the internal structure of the plan. The amount of verbal discussion of the plan also changed as the plan was created and retrieved. Creation of the sum plan showed a large amount of problem solving to create the internal structure of the plan. When the count plan was retrieved (N2), there was no verbalization within the plan.

Variability in the discussion of a solution, and in the emergence of solution pieces, is explained here as a reflection of different points along a developmental continuum. Creation of a solution occurs out from the focus of the plan to the focal line, and thence to the extension of the focal line. Retrieval of a solution shows the opposite order of emergence of plan pieces, first the extension, then the focal line. The amount of detail that is seen in subject behavior depends upon how difficult the task is for the subject. Given these two assumptions, a set of behaviors can be mapped out that reflects expertise development.

Plan creation shows a pattern of detailed, bottom-up development, from the goal, to the focus, to extensions of the focus. Once the plan has been created, it can be stored as a single chunk of knowledge, a plan schema. With further use, the internal details of the plan are compiled out of the final structure and the plan is automated (Anderson, 1982). The plan can then be retrieved and implemented in schema order, and verbal and program design will be separated (Schoenfeld, 1987). The shift from a large amount of detail to little detail, and the shift from focal to schema-based expression, can be understood as a result of knowledge compilation and automation.

To summarize, abstract program design is similar for novice and expert programmers; both groups apply abstract role knowledge to decompose a solution. Eventually, however, the problem solver needs to implement a part of the solution with a concrete plan, and at this point the two groups may diverge. If the plan can be retrieved and implemented, program design
will continue to show a pattern of forward and top-down design. If a plan cannot be retrieved, it must be created, and the design strategy will show a sequence of steps corresponding to backward and bottom-up design. Such an explanation of behavior and learning is consistent with the protocols given above, but the protocols are illustrative only. They must be related to a formal model of design and the model tested by statistical analysis before the explanation can be accepted. These two goals are achieved in the remaining sections of this article.

**DESCRIPTION OF THE MODEL**

The process of development for a plan schema shows two stages (Figure 1), separated by the plan focus. Given a problem goal, the problem solver searches for a plan to achieve the goal. If there is a plan schema that achieves the goal, it is implemented. If no schema can be found, the goal is decomposed until it can be matched with schema knowledge and implemented, creating the plan focus that directly implements the current goal. The focus is then expanded until the program plan achieves the original problem goal. The process of goal decomposition and plan recomposition creates the backward and bottom-up development of plan schemas illustrated in the protocols.

The heart of the model is the concept of a plan focus (Rist, 1986b). The focus is created at the point where a nonprogramming plan can be realized in program code, and marks the start of detailed design in the domain of the program. The remainder of the plan is then created by expanding the focus,
which in turn may require the creation and solution of new subgoals. The focus acts as an anchor for the evolving plan, since all the other parts are added so that the focus can work in the right way, to achieve the goal. It is central to the full plan because it directly implements the goal. All the other parts of the plan are based upon this first part, and added in the service of the focus.

A complication arises when the plan is too complex to be matched with a plan schema, and must be decomposed. There are two types of goal decomposition. The first type is based upon the slots in a schema, such as the total and number slots in the (nonprogramming) plan schema "average is total/number." To achieve the average goal, both these slots must be filled with detailed plans; in the development of the solution, only one slot will be considered at a time. The second type of decomposition deals with repeated events. Consider the case of reading in the rainfall per day for a single month. For the novice, there is no stored plan for reading in a large set of numbers, so the goal has to be decomposed into the simpler goal of reading in a single rainfall value. A plan for this goal is retrieved, providing a focus for problem solving. This focus must then be extended to achieve the original goal, by the addition of a loop to repeat the read plan. For the complete loop plan, the focus is the part that does the actual work (reading in a value) while the loop statement is a supporting plan for the focus.

At any point in the planning process, development can stop or go awry. If the subject does not know what an "average" is, for example, solution development stops at the plan retrieval stage. Even if the subject retrieves the nonprogramming average plan correctly, one of the slots (say "number") might drop out of memory, the plan might not be fully developed, or might be developed incorrectly. Such errors were discussed above, but will be ignored in the statistical analysis.

In the general case in complex problem solving, there is no stored plan for the current goal, but a decomposed goal can be matched to a schema. In this case, a complete focal plan for the goal, not just a focus, can be discussed. A plan is retrieved, but must then be extended and embedded in a larger structure to provide a solution to the original goal. In the Noah program, for example, the subject first encountered the need to read in a long series of numbers, the rainfall each day for a month. This goal was not directly achievable for many subjects, so it was decomposed into the simpler goal of reading in a single rainfall. There was a known plan for this, the prompt plan, which was retrieved and applied. The loop was then added to support the focus, showing forward development of the prompt plan, but bottom-up development of the loop plan.

The model of program design is based upon a very simple idea. If knowledge can be found to guide program design, top-down and forward design will be seen. The knowledge can be abstract, in the form of knowledge about role structure, or it can be specific, in the form of particular program plans.
If knowledge cannot be found, a solution must be created by focal expansion, and bottom-up and backward design will be seen. With the development of expertise, plan creation changes to plan retrieval, and plan expression changes from backward and bottom-up to forward and top-down.

**DESCRIPTION OF THE PROTOCOL STUDY**

**Subjects**
Subjects were volunteers from an introductory programming course in Pascal at Yale University. They were paid $5 per hour for their participation in the study. The course was designed as a first course in programming for students who would go on to major in computer science. Fourteen subjects started the study, but 2 withdrew from the programming course after the first session and 2 after the second session. Of the remaining 10 subjects, 7 were complete novices in programming, 2 had fewer than 20 hours experience in BASIC and 1 had about 100 hours experience in FORTRAN.

**Materials**
Eight problems were used in the study, varying over a wide range of problem types and difficulty. The simplest problem asked the subject to calculate the volume of a box-like house; the most difficult problem required 2113 weights to be sorted into ascending order. The problems were designed to be difficult or ill-defined (Simon & Hayes, 1976) for a person at the subject’s current level of knowledge. Four problems involved program loops, 4 did not. The first problem required simple input, calculations and output in the solution. The second problem required complex calculations; the third required the IF statement, the fourth needed a FOR loop, the fifth used arrays, the sixth WHILE or REPEAT loops, the seventh was a plan isomorph of the second and the last required complex loop structures. The distribution and timing of the problems within the course, and the average time taken in minutes to solve each problem, is shown in Table 6.

**Procedure**
Each problem was presented immediately after a topic had been taught in class, in an attempt to tap the subject’s declarative knowledge before it had been proceduralized. The first session was run in Week 3 of the course, and subsequent sessions each week for the next 5 weeks. No session was run in

<table>
<thead>
<tr>
<th>Problem</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week presented</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Looping</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Solution time (mins)</td>
<td>17</td>
<td>28</td>
<td>44</td>
<td>73</td>
<td>65</td>
<td>67</td>
<td>25</td>
<td>45</td>
</tr>
</tbody>
</table>
Week 5, since the lecturer taught procedures that week. Each session lasted approximately 1 hour, depending upon the subject and the problem. Several subjects were allowed extra time if they needed and desired it; a few subjects did not complete the most difficult problems.

In each session, the subject was presented with a problem and asked to write a program to solve the problem. For the first and last sessions, one simple and one hard problem were presented. The problem was written on a sheet of paper and the program was written by hand on blank paper. This produced a trace of the order of emergence of the subject's written solution. No computer was used in the study. The problem-solving process was the focus of interest, and it was suspected that use of a computer would hide some of the transient stages in solution development (self-censorship) and add an unwanted level of complexity (the computer interface) to the study.

Subjects were asked to talk aloud about what they were thinking while they were solving the problem and writing the program, and prompted if they fell silent for more than 30 seconds. Little prompting was actually required after the first two sessions. A lapel microphone was attached to the subject's collar and the whole episode recorded on videotape. The episode was later transcribed and recreated in written form from the videotapes. Each written utterance, whether it was a set of notes, a calculation, an example of a line of code, was numbered to show the order in which elements of the solution appeared. This numbering system allowed the emergence order of lines of code to be easily recorded. The numbers were then embedded in the verbal protocols, in order to recreate the whole episode. This allowed verbal discussion of a solution to be easily matched with the corresponding program code that implemented the solution.

Coding of Data

Both verbal and program protocols were used to identify solution development. The complete plan was identified from the finished program. The order of emergence of the plan pieces, both verbal and coded, was then scored for forward or backward development. During plan design, the goal (O) was usually implicit and only mentioned in the verbal protocol. The scoring of backward or forward expression was therefore based upon the other two plan components; an emergence order of CI was scored as backward expression, an emergence order of IC as forward expression. If a plan was incomplete (C or I missing), the missing role was scored as though it would appear later. All incomplete plans actually had the Initialize role missing (35, or 8.2% of the data). If the subject tried to implement the program on a computer, these roles would be added at debugging time. Since the study only scored whether the role was added before or after the focus, and missing roles had to be added eventually, the missing role was scored as following the focus.
A description of the coding scheme was written and given to two independent raters, advanced graduate students in the department of computer science at Yale. One half of the data for 3 programs (Programs 2, 4 and 8) was scored by the raters and the author. This scoring identified the ICO components of the coded plans and the verbal utterances associated with each plan. The initial, 3-way agreements are shown in Table 7. Interrater reliability, for 3 raters, was above 80% for each of the 3 programs. Conflicts in interpretation were discussed and the coding scheme amended. The author then scored all the data using the amended scheme and created the forward and backward counts from the scored data.

**RESULTS**

The development of solution plans in this study provides strong evidence for the primacy of focal expansion in problem solving. When knowledge was not available, a solution showed a pattern of backward and bottom-up development; when knowledge was available, a solution showed forward and top-down development.

Evidence for the shift from backward to forward design comes from three sources. In each set of data, a comparison is made among cases in which knowledge was not available and in cases where it was. First, the underlying model of development of a solution is that verbal discussion of a plan precedes implementation in program code. Thus, verbal protocols should show a higher level of backward expression than code protocols. Second, the first implementation of a new plan will be a single, focal line, that directly implements the goal. The preconditions of this plan will then be added, showing backward development in the code protocols. Third, backward will be replaced by forward expression as the plan schema is learned and can be retrieved as a single unit.

**Plan Creation**

To examine the behavior seen during plan creation, we may look at the first time a plan was needed, and test whether backward development was seen. For this initial problem attempt, the information to guide a solution does not exist, and must be generated. In the series of programs, there were five
plans unique to a particular program. The plans all involved a simple loop in various circumstances; one, for example, tested a number to see if it was prime, another partially sorted an array. The use of focal expansion for plan creation was easily seen from the verbal protocol data: bottom-up development was used 85% of the time. Focal expansion was also the normal strategy in the creation of code for these unique plans: it was used 72.5% of the time. There was a significant shift from bottom-up verbal creation to top-down implementation in program code \((t(10) = 2.31, p < .05)\).

Change in Plan Expression with Experience

To generalize the use of backward development from problem solving to learning, we may look at the change in plan expression over time. Five plans were repeated over the series of problems: the prompt, label, running total (count and sum), found and loop plans shown in Table 2. These five plans may be grouped into two types. The first four plans provide evidence for a shift from backward to forward expression, so the data were collapsed for these plans. The last plan, the loop, provides evidence for a shift from bottom-up to top-down expression. Only simple loops were used in this analysis, where the focus of the loop was a prompt, a label, or an initialization.

The change in plan expression with experience may be tested by comparing the degree of schema expression in plan creation versus plan retrieval. For plan creation, the first occurrence of each plan was scored and the percentage of schema expression (forward or top-down) calculated. For plan retrieval, the percentage was calculated over all the remaining occurrences for each plan. The change in expression between creation and retrieval is shown in Figure 2.

There was a significant shift from focal to schema-based design with experience \((F(1,9) = 5.14, p < .05)\). The two types of plan did not differ on their degree of schema expression \((F(1,9) = .07 \text{ n.s.})\) and there was no interaction of plan type with experience \((F(1,9) = 0.25 \text{ n.s.})\). Both lines in Figure 2 show a high level of forward expression even when the plan was created. This could be because of two factors: prior experience with the plan or prior verbal planning. Since the test times were chosen to avoid prior experience using the plan, this factor may be discounted. Only the loop plan was complex enough to allow reliable scoring of the verbal data; for the other plans, a subject would simply state the goal and code the plan.

Verbal and program data for the loop plan are shown in Table 8; the data for program code are taken from Figure 2. There was a significant increase in top-down expression from verbal planning to program code \((F(1,9) = 21.93, p < .001)\). There was also a significant increase in top-down expression from plan creation to plan retrieval \((F(1,9) = 6.30, p < .05)\). There was no interaction of the two independent variables \((F(1,9) = 1.12, \text{n.s.})\).
Plan Application
A more detailed picture of the change in plan expression over time is given by graphing the degree of forward expression for every occurrence of a plan. Graphs for the prompt and label plans are shown in Figure 3. This figure shows the expression of the plan for each program in the series; for some programs, the plan had several avatars. The graphs repeat the shift seen in the summary tables, from initial backward to subsequent forward expression. There is one interesting and important exception. The graphs of the prompt and label plans both show a drop the first time that plan had to be used in a new context, inside a loop; this point is indicated by an arrow in each graph. The levels of forward expression then recover to their previous values.

To explain this drop, the level of data analysis has to be changed in order to look at plan expression in single programs. No new model is needed to ex-
Figure 3. Graphs of prompt and label plan expressions: reversion to backward expression with new context

plain the data. The explanation for the drop lies in the interaction between plan construction and retrieval, that is, in the particular plan connections that were used in constructing a solution to the complex problem of looping. The focus of the prompt plan, which does the real work in achieving the goal, is the READ statement; it is also the focus for the original goal, the goal to read in many values. The expression of the prompt plan depends upon which goal is expanded first by the problem solver.

If the prompt plan is chosen to be expanded first, the goal to read in a single value will retrieve the plan and it will be implemented in code. The goal to read in many values will then be resumed and the loop statement added. This set of actions will be reflected in forward development of the prompt plan and backward development of the read loop plan. If the loop plan is chosen to be expanded first, the read statement will be implemented as the focus of a plan to read a single value and the loop added to repeat the read. The prompt will then be added to the read statement as an afterthought, showing backward development for both the read loop and the prompt plan. The drop in the level of forward development for the prompt plan results from the form of the plan that is retrieved to implement the goal: the read statement, or the prompt plan. The same story can be told for the drop and recovery in forward development for the label plan.

CONCLUSIONS

The model described in this article made two basic claims about the effect of knowledge on behavior. If knowledge can be found to guide program design, top-down and forward design will be seen. If knowledge cannot be
found, a solution must be created by focal expansion, and bottom-up and backward design will be seen.

These claims were supported in four ways. First, the initial attempt to create a plan was dominated by focal expansion in both the verbal and code development of the plan. Second, verbal discussion, which defines the first approach to creating a plan, showed more focal expansion than the implementation of that plan in program code. Third, plan creation changed to plan retrieval with increasing experience with a plan, and focal expansion changed to schema expansion. This shift was reflected in a change from backward and bottom-up design, to top-down and forward design. The last piece of evidence for focal expansion is especially intriguing. When a known plan had to be applied to a new and difficult problem, subjects showed a reversion to bottom-up design. It thus appears as though focal design is used when a problem is new, and a plan has to be created, or when a problem is difficult, and the problem solver has to modify or extend a known plan.

The study supports the results of previous research, but treats them as special cases of a more general model. Forward design was seen in those studies because knowledge could be applied to guide the problem-solving process. For Anderson (1982) and Visser (1987), the knowledge was in the form of specific schemas that could be applied to the problem. For the other studies, it was in the form of general knowledge about what a program looks like; first the input, then the calculation, then the output. Using this knowledge, the program could be designed by the process of schema expansion, showing a pattern of top-down and forward design. If a subject could retrieve the program plans required for a solution, top-down design could be maintained down to the level of program code. If a plan was not known, it had to be created by a process of focal expansion. Development of a plan backward from the goal, and backward and bottom-up from the plan focus, is thus basic to problem solving because it is a primitive method of solution development. It is used when the problem solver doesn’t know of anything better to do.

**DISCUSSION**

The model proposed in this article has been based upon a combination of new techniques to analyze problem solving. First, the use of both verbal and program protocols has given a more complete trace of the way a solution is generated during the process of problem solving. Second, the precise definition of plan roles has allowed a reliable measure of forward and backward development of plan pieces. Third, the method of comparative protocol analysis has created a quantitative base for the study of complex planning. Last, the use of a longitudinal study has allowed a principled definition of new and old plan schemas, and thus allowed many of the comparisons most crucial to testing the proposed model. The product of the techniques is a
detailed model of the way in which a problem solver creates and utilizes knowledge to form a solution to a complex problem.

By capturing the process of plan creation, the model of focal expansion provides a principled definition of the deep structure of a program. The deep structure consists of the detailed network of plan and goal dependencies that is required to achieve the goal defined in the problem specifications. During plan creation, the actions in a plan are added to support or enable previously defined pieces, until the focus is completely supported and can be executed to achieve the goal of the plan. The deep structure of a plan is composed of actions, connected by plan dependencies. The deep structure of a program is composed of plan schemas, connected by plan dependencies, and merged together in the correct way. Given the development of the schema, it is now possible to provide a precise answer to the question of what is in a schema, and how complex schemas are built up from simpler pieces.

The explicit definition of deep structure provides a method for adapting schemas to different situations. As a plan is developed backward from the goal, there may be several possible plans to achieve the current subgoal, and one of them will be chosen to form the next extension. When the schema is later retrieved, the current problem may be different, and a different plan will be required at that point. To give a simple example, the original read loop discussed in the paper used a FOR loop to generalize the read of a single rainfall value, so that a fixed number of values could be read for a month. In the second use of the read loop, this option was not viable because the number of letters was not known. In this case, the focus of the loop was retained, but the extension had to be changed to use a WHILE loop, and a counter then had to be added. The model of plan creation, from the focus to a series of extensions, nicely captures the relation among different versions of a schema and explains precisely where each decision was made during plan creation. In addition, it provides an explanation of program variability, by defining the decision points within the process of backward and bottom-up design.

Given that novices use focal expansion implicitly to create a solution, the model can be used to give explicit guidelines for designing a program. Recent approaches to teaching programming have emphasized the plans novices need to construct programs (Anderson & Skwarecki, 1986; Bonar & Cunningham, in press; Soloway, 1986). Attention can now be directed to teaching plan-building methods, which can be used to create both the specific schemas commonly used in program design, as well as any new or unusual schemas that may be required in a solution. Knowledge about how to build plans is particularly useful in the creation of complex schemas. As a solution gradually evolves from the initial focus to the final, complex plan, the definition of a solution by multiple steps is natural and inherent in the model of focal expansion.
This article has explained behavior in a complex domain, program design, in terms of the knowledge used by the problem solvers. It has traced the common plan schemas in the domain from the initial formation of the plan, to their storage as schemas, to their later retrieval, implementation, and extension in new and more difficult problems. Plan use changed from backward and bottom-up plan creation to forward and top-down plan retrieval. Because programming provides a general medium for plan definition and execution, plans in a program can be arbitrarily complex. It is thus feasible that the processes of schema creation, retrieval, and implementation discussed here are not limited to programming, but provide a general model of plan creation and knowledge acquisition.

Original Submission Date: October 17, 1988.

REFERENCES


