Int. J. Man–Machine Studies (1991) **35**, 517–540

The initial stage of program comprehension

SUSAN WIEDENBECK

Computer Science and Engineering Department, University of Nebraska, Lincoln, NE 68588, USA

(Received 4 October 1989 and accepted in revised form 27 July 1990)

Beacons are stereotypical segments of code which serve as typical indicators of the presence of a particular programming structure or operation. Four experiments were carried out to study the role of beacons in programmers' initial formation of knowledge of program function. It was found that the presence of a beacon made a program easier for experienced programmers to comprehend on initial study. This was found to be true even when the specific program containing the beacon was previously unfamiliar to the programmer. Also, beacons which were inappropriately placed in a program where they did not belong lead to "false comprehension" of the program's function. If a strong beacon for some operation was present, programmers tended to use it to form their initial idea of a program's function and to largely ignore other information which contradicted it. As a whole, the results of the experiments suggest that beacons may play an important role in the initial high-level comprehension of programs.

1. Introduction

When initially faced with an unfamiliar program, how do programmers gain an overall high-level comprehension of the goals of the program and the operations used to implement them? Program comprehension is a gradual process of assimilation through study, and some kind of program overview appears to be a first pass which programmers make in attempting to comprehend. For example, Jeffries (1982) discovered that expert programmers initially scan a program listing in some systematic way to pick up orienting information, before they apply themselves to a debugging task. After gaining such an overview, a programmer clearly will not be able to describe the details of the program's operations, control flow, or data flow with any great depth or accuracy. However, the orienting phase is important because it allows the programmer quickly to develop a sort of mental map of the program which, though incomplete, includes the basic goals and operations and can be used later to focus the more detailed study of the program, which is necessary for debugging or modification.

One effective way to acquire orienting information would be from the documentation. A useful overview could be contained in a flowchart or structure diagram, in overall descriptions of the program's purpose and operations, or in the somewhat more specific documentation of the goals and implementation of individual modules. However, there appear to be two problems with using documentation for orienting information. One is that in the world programs differ greatly in the amount, level of detail, and quality of their documentation. Documentation which is non-existent, too sparse, too detailed, or confusing fails to aid the programmer trying to develop an accurate overview. A second problem is that programmers do seem to have a

0020-7373/91/100517 + 24$03.00/0

tendency to go to the code itself to develop an initial grasp of the program. Early code scanning was reported by Jeffries (1982) and was also observed by us in a pilot study where we tried to provide useful overview documentation.

In this paper we present a series of studies of programmers gaining a high-level comprehension of programs from the code itself. The focus of the studies is on investigating the role in comprehension of beacons, or key features, of the code. As will be seen, the studies suggest the existence of a connection between the presence of a beacon and superior program comprehension.

The first section below describes beacons and their relation to other posited knowledge structures in program comprehension. It then presents the case for the existence and use of beacons in program comprehension, including a summary of the results of our previous research. The following four sections report on the four experiments which we did to study the effect of beacons on program comprehension. The concluding section evaluates our current state of knowledge about the early phases of program comprehension.

2. Beacons in program comprehension

Brooks (1983) proposed the concept of beacons as a knowledge structure used in computer program comprehension. His theory of program comprehension is strongly top-down and hypothesis-driven. According to the theory, the programmer does not study a program line-by-line but rather forms a series of increasingly specific hypotheses about program function. The highest-level hypothesis concerns overall program function and may be derived from documentation or from more scanty sources, such as a program name. This overall hypothesis leads the programmer to expect to see certain objects and operations in the program, and these expectations then form a second level of more specific hypotheses about the program. The programmer's job during comprehension is to verify the hypotheses from information in the program text, modifying or rejecting hypotheses which are not supported, until reaching an adequate comprehension of the program for the purpose at hand.

The process of verifying hypotheses is central to Brooks' theory because it is the point where the top-down hypothesis formation activity must link up with the data of the program text. Brooks believes that programmers verify hypotheses not by a minute study or simulation of every line of the code, but by searching for beacons, which are key features that typically indicate the presence of a particular structure or operation. Thus, beacons can be described as idiomatic or stereotypical elements in program code. An example of some beacons given by Brooks is that in a master file update program one expects to find two sources of input, possibly a sort routine, and definitely a merge routine. While the idiomatic beacons may not be present in every implementation of a given operation, they occur frequently enough to be strongly associated with the operation for the experienced programmer. If the experienced programmer searches a program and does not find the expected beacons, he or she will search more thoroughly. If this search, too, fails, the hypothesis will be weakened, perhaps revised, or even discarded.

Beacons were described by Brooks in the context of his top-down theory of program comprehension. However, the search for beacons is also consistent with theories which contain both top-down and bottom-up elements (Littman *et al.*, 1986).

In such a theory the programmer begins by assigning meaning to lines or groups of lines in the program text. This information is stored in the programmer's memory and new information about program function is assimilated to it incrementally as more of the program is studied. At some stage in this process the programmer has enough information to make top-down hypotheses about program function. These hypotheses may then be verified by a search for beacons.

Beacons are related to the concept of programming plans developed by Soloway and his colleagues (Soloway *et al.*, 1982; Ehrlich & Soloway, 1984; Soloway & Ehrlich, 1984). Basic plans are stereotyped program fragments for accomplishing a single simple goal, for example, summing a series of numbers or guarding against reading bad input data. A program can be thought of as a more complex plan which joins several simple plans together in some way to achieve a more complex goal. In a complex plan there will often be some key part which carries out the central action, for instance, the test and swap in a sort, which is central because it is the prime operation which moves the problem toward the goal of a sorted list. This key part has been referred to as a "kernel idea" (Kant, 1985) or a "focal segment" (Rist, 1986). The kernel idea or focal segment in a program would serve as a beacon in program comprehension because it is central to the goal of the program and occurs in many variations of it (e.g. the test and swap occur in many kinds of sorts). However, while the kernel or focal segments of complex plans would always serve as beacons, beacons often include more than just that. They are any surface feature of a program which strongly points to the program's function. In addition to kernel or focal lines, this might in some cases include the control structure of the program (such as the looping structure in a sort), procedure names, and variable names.

The implementation cliché, a concept closely related to beacons and plans, has been used by Rich & Waters (1988) in the development of their Programmer's Apprentice. Rich & Waters argue that programmers think mostly in terms of commonly used combinations of elements, or clichés, rather than in terms of primitive elements like assignments or comparisons. In the Programmer's Apprentice they have created an automated assistant which stores libraries of such clichés which can be used in program implementation, design, and requirements specification. For example, in program synthesis by inspection clichés in the specifications are recognized and implementation is carried out by choosing among various stored implementation clichés. Presumably programmers analyse programs by inspection, too, recognizing clichés and calling to mind their properties. If this is true, then programs written with the aid of such an apprentice should be easier to understand because they share a common vocabularly with the programmer who later tries to comprehend them. Likewise, automated aids to program comprehension might be built which could recognize the occurrences of implementation clichés and use that information to make hypotheses about program function. Along these lines, a plan-based programming tutor has already been built which analyses and understands elementary student programs (Johnson & Soloway, 1985).

Previously we carried out experiments which suggested an association between programmer expertise, recognition of a beacon, and comprehension of programs. In one study (Wiedenbeck, 1986*a*), we found that experienced programmers recalled significantly more of the beacon lines in a program than did novices, 79% for experienced programmers, as opposed to 14% for novices. The recall of non-beacon lines was not significantly different for the two groups. At the same time,

experienced programmers were better able to answer comprehension questions about the program. In a second experiment experienced subjects studied a program for comprehension but did not memorize it (Wiedenbeck, 1986*b*). They were then given one program line as a cue and were asked to reconstruct from memory the surrounding program lines. One of the parts to be reconstructed was the beacon and the other six parts were non-beacons. Subjects were told to reconstruct the seven segments starting with the one they were most certain about and going to the least certain. Subjects reconstructed the beacon segment much more accurately than any of the non-beacon segments (88% correct for the beacon and from 4% to 29% correct for the other six segments). In addition, the beacon segment was most often completed first by subjects, indicating their greater confidence in reconstructing it.

Our earlier experiments only established an association between programming expertise, memory for a beacon, and program comprehension (higher expertise being associated with better memory for the beacon and also with better program comprehension). The experiments reported below elaborate on the previous studies. They compare programmers' comprehension of beacon-containing and non-beacon containing programs and also look at the effect of a beacon occurring in anomalous surroundings.

3. Disguise experiment

The purpose of this experiment was to determine whether a program containing a beacon for a well-known operation would be understood better than a very similar program, accomplishing the same thing, but not containing the beacon. This experiment was meant to extend our previous research, which had not directly compared a beacon-containing with a non-beacon-containing program. In this experiment all subjects were expected to comprehend the beacon-containing program better. However, experienced programmers were expected to be more strongly aided by the presence of a beacon, based on their performance in our previous research.

3.1. SUBJECTS

Subjects at two levels of expertise participated in the experiment. The two levels of expertise are referred to here as novice and advanced. The novices were mostly sophomores at the University of Nebraska at Lincoln or the University of Nebraska at Omaha. Many, though not all, were computer science majors or prospective majors. They had taken an average of 3·4 university courses involving programming. This usually included a two course computer science introductory sequence and a third course, either assembly language or a data structures and algorithms course. Six of them had some part-time work experience in programming. The advanced group consisted of graduate students in computer science at the University of Nebraska at Lincoln and working programmers. (There is some overlap in these classifications since some working programmers were also in computer science degree programs.) The advanced group, though diverse, was distinctly more experienced than the novices by the background measures collected. Subjects in the advanced group had taken an average of 10·5 programming-related courses. Two-thirds had part-time programming experience, and one-third were working or had previously worked full-time as programmers.

3.2. MATERIALS

Two versions of a Pascal procedure were used for testing in this experiment. One of them, referred to here as the "no-disguise" program, was a standard textbook version of a Shellsort (Figure 1*a*). We chose a Shellsort program because we felt that the domain of sorting would be familiar to all participants in the experiment. Moreover, a sorting program seemed to be a good choice because the swap in a sort is an almost ideal instance of a beacon: it is typically present and is the prime operator that moves the problem toward the goal of a sorted list. The program used here contained 25 lines and occupied about half of a printed page. The program was well-spaced and indented but did not contain any comments. Most importantly it contained the beacon lines which perform the swap of values in the array ($t:=a[j]$; $a[j]:=a[k]$; $a[k]:=t$). In this version of the Shellsort the swap occurs about two-thirds of the way through the program. Thought was given to what were the limits of the beacon. Did it contain just the swap itself or the comparison (line 13, Figure 1*a*) plus the swap? Our previous work had shown high recall of the swap lines as well as a marked cohesiveness in memory (i.e. a subject who recalled one of them usually recalled all three). This was less true when one included the comparison line. It was recalled less frequently than the swap lines and seemed less cohesive with them, so a decision was made to treat the swap alone as the program beacon.

The second version of the Pascal procedure used for testing, referred to as the

```
 1  Procedure P1 (var a: atype; n: integer);
 2  var i, j, k, m: integer;
 3    t: char;
 4  begin
 5    m:=n div 2;
 6    while m>0 do begin
 7      i:=m;
 8      repeat
 9        i:=i+1;
10        j:=i-m;
11        while j>0 do begin
12          k:=j+m;
13          if a[j]>a[k] then begin
14            t:=a[j];
15            a[j]:=a[k];
16            a[k]:=t;
17            j:=j-m
18          end
19          else
20            j:=0
21        end;
22      until i=n;
23      m:=m div 2
24    end
25  end;
```

(*a*)

```
 1  Procedure P21 (var a, b: atype; n: integer);
 2  var i, j, k, m: integer;
 3  begin
 4    m:=n div 2;
 5    b:=a;
 6    while m>0 do begin
 7      i:=m;
 8      repeat
 9        i:=i+1;
10        j:=i-m;
11        while j>0 do begin
12          k:=j+m;
13          if a[j]>a[k] then begin
14            b[j]:=a[k];
15            b[k]:=a[j];
16            a:=b;
17            j:=j-m
18          end
19          else
20            j:=0
21        end;
22      until i=n;
23      m:=m div 2
24    end
25  end;
```

(*b*)

FIGURE 1. (*a*) Beacon disguise experiment: no-disguise program. (*b*) Beacon disguise experiment: disguise program.

"disguise" version, was very similar to the first in most ways (Figure 1*b*). It was also a Shellsort, 25 lines long, with standard page layout and no comments. Variable names were the same as in the previous version except for one variable which was used in the previous version but not used in this version and another new variable which was added here. The overall number of variables remained the same. A brief look at the two program versions convinces one of their close similarity in overall appearance and structure.

The key difference between the two programs is that the second one disguises the swap beacon. A swap is still done, but not in the normal way where a simple scalar variable is used as a temporary variable during the swap (*t* in the no-disguise version of the program). Instead, a second array called *b* is introduced and used for swapping. As in the previous version, swapping is done in lines 14–16. The contents of *a* (the array to be sorted) are copied into array *b* at the beginning of the program. Then, when a swap of two elements is necessary, it is done by copying values from array *a* (where they are in the wrong position) into the right locations of array *b* (lines 14 and 15). For example, to swap *a*[3] and *a*[5] one copies the value in *a*[3] into *b*[5] and the value in *a*[5] into *b*[3]. Since the other values in *b* are already identical to those in *a* (from copying the whole *a* array earlier), *b* now contains the order of values we want, and the whole array *b* is copied back to array *a* before the next iteration begins (line 16). This is obviously an inefficient way to accomplish the swap and would not be done in a real program. However, it fulfills the purpose here of creating a second program version, almost identical to the first, carrying out exactly the same function, but with the key lines disguised.

3.3. PROCEDURE

For each level of expertise, 20 subjects worked on the no-disguise version of the program and 20 worked on the disguise version (80 subjects in total). The subjects were run in small groups. They were given the program for 3 min and were told to study it primarily for comprehension. They were also told to try to remember as much of the program as they could, but it was emphasized that this was a secondary task. At the end of the study period, subjects carried out three tasks: (1) they demonstrated their comprehension by describing the program's function (2 min allowed); (2) they rated their confidence in whether they had correctly understood the function (1 min); and (3) they recalled the program (4·5 min). A pilot test had shown that these times were adequate for even the slower novice subjects to finish the tasks.

The main point of interest in this experiment was whether subjects understood overall program function better in the version containing the beacon. The confidence measure was included as a more subjective supplement to the program function measure, since subjects might not differ in their ability to state the function of the two versions, but still might differ strongly in how confident they were about their conclusions. The recall measure was included to see whether the swap was recalled better when it was in its prototypical form than when disguised.

3.4. RESULTS

The results on the program function and recall measures are described in turn in this section. The confidence ratings are not reported because to analyse them required

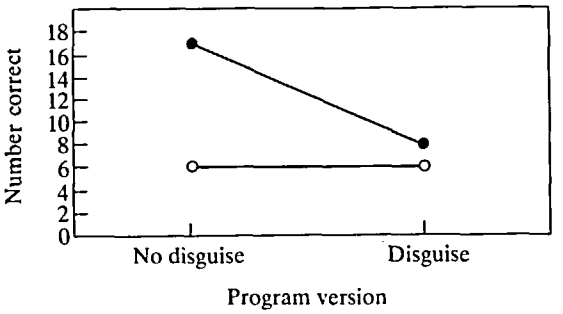

FIGURE 2. Disguise experiment: number of subjects giving correct program function ($N = 80$). ●, advanced; O, novice.

doing separate comparisons for subjects who correctly stated the function of the program and subjects who did not. However, when analysed with regard to the correctness of the solution, the number of subjects in some categories was too small to meet the requirements of a Chi-square test, and it was not feasible to increase the sample size beyond 80.

For determining program function, subjects were considered correct if they said that the program was a sort, even if they did not say what kind of sort. Saying that the program performed swapping, was not considered correct. The number of subjects who correctly stated the function is given in Figure 2; broken down by level of expertise and program version. A $2 \times 2 \times 2$ Chi-square test was run on the data with level of expertise as one factor, program version as a second factor, and the correctness of the subject's response as the third factor. It showed that advanced subjects were more correct in determining the function than novices ($\chi^2 = 6{\cdot}05$, d.f. $= 1$, $p < 0{\cdot}05$). Also, the no-disguise version was correctly understood significantly more often than the disguise version ($\chi^2 = 6{\cdot}05$, d.f. $= 1$, $p < 0{\cdot}05$). The interaction of expertise and program version was not significant.

In program recall our greatest interest was in determining whether the beacon-like swap lines in the no-disguise version were recalled better than the non-beacon-like swap lines in the disguise version. Both versions had three lines making up the swap. Also, in both versions the swap appeared at lines 14 through 16. In scoring recall, for a line to be considered correct it has to be recalled exactly except for indentation, spacing, capitalization, and end-of-line punctuation. To analyse recall of the swap in the two versions, a two-way ANOVA was run with expertise and program version as the independent variables and recall of the swap (lines 14–16) as the dependent variable. Since there were three lines involved in the swap in each version, the highest possible score a subject could get was three.

The average recall scores are shown in Figure 3. We found that, while there was a trend for subjects to recall the swap lines in the no-disguise version better than the swap lines in the disguise version, none of the differences was significant. Another analysis of recall of the entire programs also found no significant differences.

3.5. DISCUSSION

The principal result of this experiment was that a beacon-containing program was understood correctly more often than a non-beacon-containing program. The

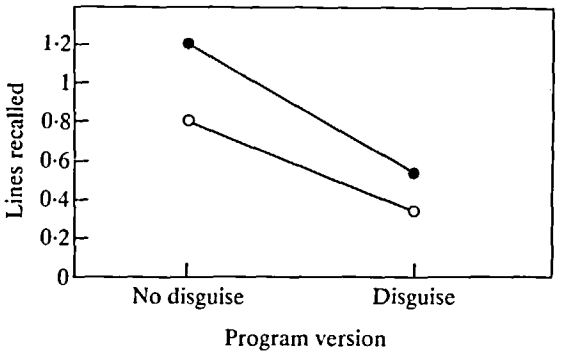

FIGURE 3. Disguise experiment: mean recall of 3 swap lines. ●, advanced; O, novice.

program versions used in the experiment were virtually identical except for the swap. The better comprehension of the no-disguise version containing the typical, or beacon, swap suggests that those lines have special importance in program comprehension. Without them present in their typical form, it was more difficult to determine the function of the program during a quick, initial study of it. The finding that advanced programmers were better than novices at determining program function, regardless of the program's form, is in accord with most past research on programming, including our previous work on beacons. The result was expected in this experiment because the advanced group was considerably more experienced than the novice group by the measures of expertise collected. However, we did not find, as we had expected to, that advanced subjects were helped significantly more by the presence of a beacon in comprehending the program that novices were. Since programmers must learn to associate a beacon with a given program type through experience, it seemed sensible that advanced programmers would take advantage of them more in program comprehension. The failure to see this result may mean that the association of the swap with sorting is a strong one which students learn early in their experience with sorting algorithms.

The negative results of the recall of the beacon lines was surprising. One would expect the swap lines in the no-disguise version to be recalled better, at least by advanced programmers, because of their idiomatic nature. From Figure 3 it appears that the pattern of results for the advanced programmers was the same as in the function measure, although the differences were not large enough to be statistically significant. Also, they were not nearly as large as in Wiedenbeck (1986*b*), where advanced programmers recalled an average of 78% of a standard non-disguised swap beacon. Here they recalled only 33%. The reason for the discrepancy between this and that previous work is not clear. One possibility is that the directions to subjects in this experiment emphasized determining function more and recall much less than in the previous experiment.

The results of this experiment support a connection between the presence of a beacon and comprehension of program function (though not program memorability). However, the results must be viewed with caution because of a potential confound: the possible familiarity of the Shellsort to the subject population. Some may have programmed it, making the no-disguise version more strongly and

immediately recognizable to them, and probably many were generally familiar with the method from their previous studies. Also, to create a disguised version of the Shellsort not containing a conventional swap, it was necessary to add a few lines of code that would not usually occur in a sorting program (for example, copying one array to another). This may have created some sort of anti-familiarity effect in the subjects, possibly even leading to wrong hypotheses. Because of these possible confounding factors, several more experiments were carried out which approached the study of beacons in different ways. Our hope was to find converging results which would clarify the interpretation of this experiment.

4. Prototypical and non-prototypical sort experiment

The previous experiment gives support to the idea that beacons aid in program comprehension. However, because of the potential confound with familiarity we wanted to run another experiment which would approach the question of the role of the beacon in a different way. If beacons are closely tied to the meaning of a program and play an important role in program comprehension, they should help not just in familiar programs but more importantly in programs never seen before. Thus, the primary purpose of this experiment was to assess the influence of beacons on comprehension of a program known to be unfamiliar to the programmer. If beacons facilitate understanding unfamiliar programs, this should become clear by comparing the comprehension of an unfamiliar program containing a beacon with an unfamiliar program not containing a beacon.

Beacons are a kind of idiom in programming. A second purpose of this experiment was to test whether all or only some programming idioms have a special effect on comprehension and, therefore, qualify as beacons. It seems that certain idioms may have an especially close connection to the meaning of a program, as we have argued the swap in a sort does. The presence of such beacons, which carry a lot of meaning and seem to be diagnostic of program function, should correlate with comprehension. Other programming idioms are more general. They occur in a wider variety of programs and, although very well known to programmers, are less uniquely associated with any one program. As a result, such idioms should be highly recognizable but will probably not correlate highly with comprehension.

4.1. SUBJECTS

Seventy-six subjects participated in the experiment, none of whom had participated in the previous experiment. Half were novices and half advanced programmers. The programming experience of the novices was somewhat less than that of novices in the previous experiment. The novices had taken an average of 2·3 programming courses and only two had any part-time work experience. The advanced subjects were similar in background to those in the previous experiment.

4.2. MATERIALS

Two Pascal procedures were used as stimuli in this experiment. To remain in the same domain as the previous experiment, both were sorting procedures. For testing the experimental question we needed two unfamiliar sorting procedures, one of

```
1  Procedure P3 (var a: atype; n: integer);
2  var i, j, k, t: integer;
3  begin
4    k:=1
5    for i:=1 to n-1 do begin
6      j:=k;
7      while j<n do begin
8        if a[j]>a[j+1] then begin
9          t:=a[j];
10         a[j]:=a[j+1];
11         a[j+1]:=t;
12       end
13       j:=j+2
14     end;
15   k:=k+1
16 end
17 end;
```

(a)

```
1  Procedure P4 (var a: atype; n, m: integer);
2  var i, j: integer;
3    b, count: atype;
4  begin
5    for j:=0 to m-1 do
6      count[j]:=0;
7    for i:=0 to n do
8      count[a[i]]:=count[a[i]]+1;
9    for j:=1 to m-1 do
10     count[j]:=count[j-1]+count[j];
11   for i:=n down to 1 do begin
12     b[count[a[i]]]:=a[i];
13     count[a[i]]:=count[a[i]]-1
14   end;
15   for i:=1 to n do
16     a[i]:=b[i]
17 end;
```

(b)

FIGURE 4. (*a*) Prototypical and non-prototypical sort experiment: protypical program (odd–even transposition sort). (*b*) Prototyypical and non-prototypical sort experiment: non-prototypical program (distribution sort).

them prototypical in that it contained a swap and the other non-prototypical in not containing a swap *per se*. The two sorting procedures chosen were the odd–even transposition sort and the distribution sort, shown in Figures 4*a* and 4*b*. Both of these sorts would rarely be used in practice. Knuth (1973) points out that the odd–even transposition sort is unlikely to be used because it is more complicated and even somewhat less efficient than other elementary sorting procedures which also have a complexity of $O(N^2)$. The distribution sort, on the other hand, is efficient, but it is a special purpose sort, since it is meant for sorting sequences of data which are fairly continuous. It was felt that both of these sorting procedures would be unfamiliar to potential subjects. To confirm this assumption a class of advanced computer science majors at the University of Nebraska was presented with each program, along with its name and a description of how it worked. They were asked whether they had any previous familiarity with the program. For each program, over 90% of the students had no previous familarity, so the programs were felt to be matched on familiarity. In the odd–even distribution sort (referred to as the prototypical sort here) the swap beacon is located at lines 9 through 11 and is buried within nested loops. It differs from the usual elementary sorting procedures in its loop incrementing and sequence of comparisons. The distribution sort (referred to here as the non-prototypical sort) uses two arrays and a series of loops, but contains no swap beacon. The idea behind the distribution sort is to count the number of keys with each value, then use the counts to move the records into position in another array on a second pass through the file. The first FOR loop in the non-prototypical sort is an initialization loop which sets the array called Count to 0. This is a programming idiom not strongly and uniquely associated with sorting, which should be recognizable and memorable as an initialization loop, but should not lead to better overall comprehension of the program.

Both programs were 17 lines long. Also, both programs were formatted to occupy the same amount of space on a printed page (about one third of a page). The programs were indented. The number of variables in the prototypical sort was six, and the number in the non-prototypical sort was seven. No documentation was given for either program.

4.3. PROCEDURE

For each level of expertise, 19 subjects worked on the prototypical sort and 19 on the non-prototypical sort (76 subjects in all). The procedure followed was the same as for the previous experiment. Subjects first studied the program for comprehension for 3 min, then were given 2 min to describe the program's function and 4·5 min to recall the program as precisely as possible. A confidence measure was also collected but was not analysed for the reasons explained in the previous experiment.

4.4. RESULTS

With respect to program function, it was felt that the prototypical sort, containing the idiomatic swap beacon, would be understood much better. This proved true. The comprehension scores are given in Figure 5 broken down by expertise and program (prototypical or non-prototypical). Even though both programs were equally new to the subjects (not introduced in classes and seldom used in practice), subjects were superior on the prototypical program. A $2 \times 2 \times 2$ Chi-square test was run with level of expertise as of factor, program as a second factor, and the correctness of the subject's response as the third factor. It showed that the difference between the prototypical and non-prototypical programs was statistically significant ($\chi^2 = 23{\cdot}38$, d.f. $= 1$, $p < 0{\cdot}05$). Likewise, the expertise factor was significant ($\chi^2 = 5{\cdot}85$, d.f. $= 1$, $p < 0{\cdot}05$), with advanced subjects performing better than novices. There was a marginal three-way interaction between expertise, program, and correctness ($\chi^2 = 3{\cdot}74$, d.f. $= 1$, $p < 0{\cdot}10$). Inspection of Figure 5 indicates that, while both novice and advanced subjects performed better on the prototypical as opposed to the non-prototypical sort, advanced subjects achieved especially high scores on the prototypical sort. In fact, the main effect of expertise is mostly explained by the advanced subjects' superior performance on the prototypical sort.

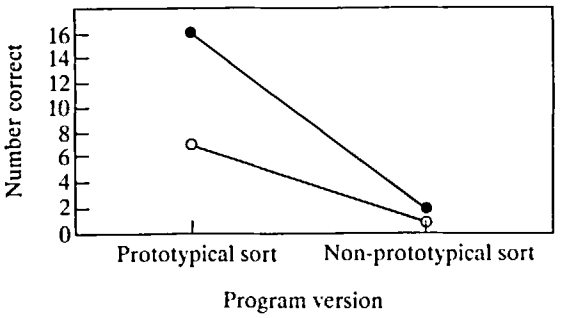

FIGURE 5. Prototypical and non-prototypical sort experiment: number of subjects giving correct program function ($N = 76$). ●, advanced; ○, novice.

A program recall measure was included to see how memorable an unknown but prototypical sort containing a beacon would be compared to a non-prototypical sort, also unknown, but not containing a familiar beacon. We compared total lines recalled in the two programs. The means for total lines recalled (out of 17) are given in Figure 6*a*. A two-way ANOVA was run with expertise and program as the independent variables and number of lines recalled as the dependent variable. We found that advanced subjects recalled more than novices, an average of 8·6 as opposed to 6·3 lines ($F(1, 72) = 6{\cdot}77$, $p = 0{\cdot}011$). Also, subjects recalled the prototypical program better than the non-prototypical program, an average of 8·8 lines recalled *vs* 6·1 lines ($F(1, 72) = 9{\cdot}49$, $p = 0{\cdot}003$). The interaction of expertise and program version was also significant ($F(1, 72) = 4{\cdot}77$, $p = 0{\cdot}032$). This reflected the fact that advanced subjects had similar levels of recall to novices on the non-prototypical sort (about six lines) but, unlike novices, recalled more of the prototypical sort (about 11 lines). To confirm the recall results we also ran an ANOVA on recall of only lines 9–11 of the two programs. In the prototypical sort program lines 9–11 were the swap beacon. In the non-prototypical sort program they had no special standing (since the program did not contain the beacon), but were simply used for comparison, since their placement corresponded to that of the swap in the prototypical program. The recall means are shown in Figure 6*b*. We found results very similar to the results for recall of all lines. There were significant differences ($p < 0{\cdot}05$) based on expertise (advanced subjects superior), program (prototypical sort superior) and a significant interaction (advanced subjects achieving especially high recall of the prototypical version).

In the non-prototypical sort there was no swap beacon, but one programming idiom was nevertheless present, an initialization loop (lines 5 and 6 in Figure 4*b*). We were interested in seeing whether this idiom was well remembered compared to the rest of the program and also whether memory for the idiom correlated strongly with comprehension of the function of the program. Considering all subjects as a single group, it turned out that only slightly more of the initialization lines were recalled (48·7%), than lines in the remainder of the program (40·5%). The novices actually recalled 65·8% of the initialization lines, showing that they were very familiar with this simple idiom. Advanced subjects recalled 31·6% of the initialization lines, perhaps indicating that they understood it more quickly, concentrated less of their study time on it, and therefore made more errors of detail in recalling it.

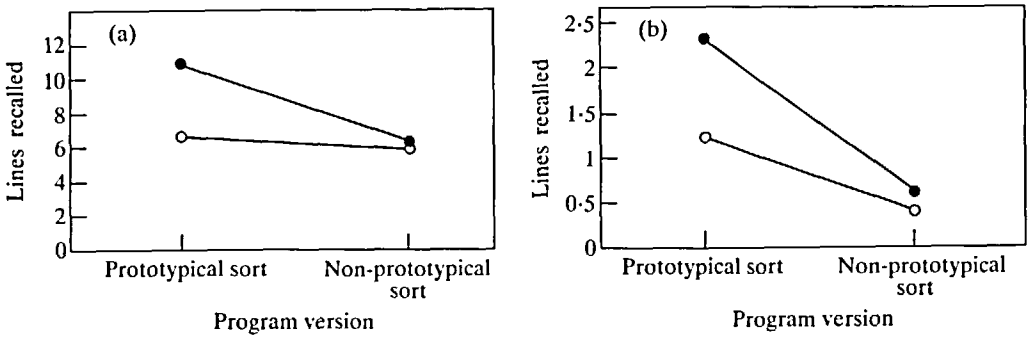

FIGURE 6. (*a*) Prototypical and non-prototypical sort experiment: mean recall of all 17 lines, (*b*) mean recall of lines 9–11. ●, advanced; ○, novice.

4.5. DISCUSSION

The most salient result of this experiment was that, given a program which they have never seen before, programmers were much more successful at comprehending if if there was a beacon present. This result supports the characterization of the beacon as a key to program understanding. Few subjects were able to determine the function of the non-prototypical program in the short time given for study. It appears that, lacking a strong sort beacon, subjects could not correctly guess the function from scanning or brief study alone. There is nothing intrinsically difficult or confusing about the non-prototypical sort. Many subjects (especially advanced subjects) recognized and described the individual parts of the algorithm, such as the initialization, counting, and copying subfunctions. However, they failed to tie the whole thing together as a sort, lacking the idiomatic beacon. Apparently deeper study and/or simulation is necessary when idioms are not used. Advanced programmers in this experiment were able to take advantage of the beacon to improve program comprehension better than could novices. Beacons are learned patterns, knowledge of which is built up through experience. For instance, the frequent appearance of a swap in different sorting programs is learned from exposure to different sorting algorithms. Advanced programmers have had much more opportunity to see many different instances of a class of programs and to associate a beacon with them. The results showed that not all idioms are beacons which lead to better program comprehension. Patterns like an initialization, which are associated with many diverse kinds of programs, do not aid in initial program comprehension because they are not associated uniquely enough with a certain kind of program.

Our results also indicated that a program containing a recognizable beacon is recalled better than one without a beacon, especially by more experienced programmers. Even though the specific programs used here had never actually been seen before, the program which contained a known beacon seemed to be more memorable. As expected, advanced programmers recalled more than novices. This result was expected based on most previous work using recall as an indicator. In this experiment the novices had somewhat less programming background than had the novices in the previous beacon disguise experiment. This may explains why an overall difference between novices and advanced subjects was found here but not in the previous experiment.

In this experiment we wanted to compare beacon-containing and non-beacon containing programs which were both correct, which were both equally unfamiliar to the subject population, and which did not contain any obvious inefficiencies which might possibly confuse or distract subjects (such as the inefficient use of a temporary array for sorting in the previous experiment). As a result, we used two distinctly different sorting procedures. Since the two programs in this experiment used fundamentally different algorithms there is a potentially confounding factor here in the algorithm. It is plausible that other features besides the swap beacon contributed to superior compreheneion of the prototypical program. However, we have some reason to believe that the swap beacon is the key element. First of all, in Experiment 1, where the two programs differed only in the swap, subjects recognized the program's function better when the beacon was present. Furthermore, the fourth experiment reported below investigated how other features of the

program contribute to or modify the interpretation of the beacon. As will be seen, we found that during a brief initial study the beacon's presence pointed strongly to the function, independent of other lines in the program. This tends to support the conclusion that the presence or absence of the beacon made the difference in this experiment. Thus, this experiment, is not definitive. However, it provides an accumulation of evidence, along with the previous experiment, for a role of the beacon in comprehension. The facilitation of program comprehension was found in the previous experiment, where the programs differed only in the beacon, but the disguised version was rather awkward, as well as here, where both algorithms were written in their standard form but there were other differences between them in addition to the presence or absence of a beacon.

5. Beacon intrusion experiment

The previous two experiments show that the presence of a known beacon aids in the comprehension of a program which contains it and that the disguise or absence of such a beacon delays, or possibly even disrupts, the comprehension process. The point of this experiment was to investigate how a beacon influences program comprehension when it is inappropriately placed in a program where it does *not* belong. Does the presence of such an inappropriate beacon lead to false recognition of a program's function? Since beacons have the power to aid program comprehension when they appear in appropriate contexts, it seems likely that they have the power to depress program comprehension, and even lead to false "comprehension," when they are used inappropriately. The experiment was designed to test this hypothesis. If false recognition of program function does occur, it will be further evidence of programmers' reliance on beacons in the initial stages of program comprehension.

5.1. SUBJECTS

Seventy-six subjects partcipated in the experiment. Half the subjects were novices and half were advanced programmers. None of the subjects had participated in either of the previous two experiments. The subjects' programming experience was the same as in Experiment 1.

5.2. MATERIALS

The experimental materials consisted of two Pascal procedures. For testing the hypothesis that a beacon inappropriately placed in a program would depress program comprehension or lead to false recognition of the program's function, a binary search procedure was used. The binary search was chosen because it, like the sort, is a common procedure which was familiar to the whole subject population. The two programs used are reproduced in Figures 7*a* and 7*b*. The first version is a standard binary search in which the file being searched is repetitively divided in half and the item sought is compared to the item at the midpoint of the file until the desired item is found or until the first and last item pointers cross. In this version, the desired item is printed out if found, otherwise a return message is printed. The second version of the program contains the beacon intrusion. The basic binary

```
 1  Procedure P5
      (var a: atype; f, g: integer);
 2  var i, j, k: integer;
 3  begin
 4    i:=1;
 5    j:=f;
 6    repeat
 7      k:=(i+j) div 2;
 8      if g<a[k] then
 9        j:=k-1
10      else
11        i:=k+1;
12    until (g=a[k]) or (i>j);
13    if g=a[k] then
14      writeln (g)
15    else
16      writeln ('return')
17  end
```

(a)

```
 1  Procedure P6
      (var a: atype; f, g: integer);
 2  var i, j, k, t: integer;
 3  begin
 4    i:=1;
 5    j:=f;
 6    repeat
 7      k:=(i+j) div 2;
 8      if g<a[k] then begin
 9        t:=a[k];
10        a[k]:=a[k+1];
11        a[k+1]:=t;
12        j:=k-1
13      end
14      else
15        i:=k+1;
16    until(g=a[k]) or (i>j)
17  end;
```

(b)

FIGURE 7. (a) Beacon intrusion experiment: no-intrusion program. (b) Beacon intrusion experiment: intrusion program.

search program is the same as in the previous version. However, here a swap beacon is inserted inappropriately into the program ($t:=a[k]$; $a[k]:=a[k+1]$, $a[k+1]:=t$). The swap, although unnecessary and inappropriate, does not affect the outcome of the search. The swap switches two elements in the second half of the file if the key sought is located in the first half and the two items are out of order. In fact, given that a binary search starts with a sorted file, this should never be true. Therefore, the swap is harmless and the binary search will execute as in the previous case.

The no-intrusion and intrusion versions of the program were matched in so far as possible. Each version was 17 lines long and occupied about one third of a printed page. Because of the swap, the intrusion version would have been longer than the no-intrusion version. To equalize the lengths, the output lines were added to the no-intrusion version. The programs were indented but contained no documentation.

5.3. PROCEDURE

The experimental procedure was exactly the same as in the previous experiments. Within each level of expertise, half the subjects worked on the no-intrusion version of the program and half worked on the intrusion version. No subject saw both versions. The same measures of performance were collected.

5.4. RESULTS

In this experiment the main interest was in determining how the presence or absence of the intruding swap beacon affected the comprehension of the basic binary search program. To measure this, a count of how many subjects said each version of the program was a binary search and how many said each version was a sort was made. If the beacon intrusion strongly influences subjects' comprehension, then fewer

subjects should say that the intrusion version is a binary search and more should say that it is a sort. The data on how many subjects understood the program as a binary search are shown in Figure 8*a*. A 2×2 Chi-square test was run. One factor was program version (no-intrusion/intrusion) and the other was whether or not the subject said the program was a binary search. The two levels of expertise were collapsed in this analysis in order to have large enough expected frequencies to do the Chi-square test. Therefore, no statistical results are reported for level of expertise. The analysis showed that performance differed marginally based on program version with the no-intrusion version being comprehended as a binary search more often than the intrusion version containing the swap ($\chi^2 = 3{\cdot}6$, d.f. = 1, $p < 0{\cdot}10$). As a second comprehension measure, we also analysed how many subjects were misled into comprehending each program as a sort. The results for this measure of performance are shown in Figure 8*b*. A 2×2×2 Chi-square test showed that there was a significant difference based on the program type ($\chi^2 = 19{\cdot}02$, d.f. = 1, $p < 0{\cdot}05$). There was no significant difference based on level of expertise and no significant three-way interaction. As is apparent, the intrusion version was much more likely to be understood as a sort than was the no-intrusion version. In fact, 50% of the group which saw the intrusion program judged it to be a sort, but only 5% of the group which saw the no-intrusion program. It is notable that almost as many advanced subjects as novices were misled into comprehending the intrusion program as a sort in the short time that they had to study it.

Finally, recall of the programs was compared. As in the previous two experiments, a line was judged correct if it was recalled exactly except for indentation, spacing, capitalization and end of line punctuation. We analysed total lines recalled using a two-way ANOVA with level of expertise and program version as the independent variables and number of lines recalled as the dependent variable. The means are shown in Figure 9*a*. The only significant difference was based on expertise ($F(1, 72) = 13{\cdot}06$, $p = 0{\cdot}001$). A second recall analysis was done which used an ANOVA to compare program lines 9–11 of the intrusion version, which contained the swap beacon, with lines 9–11 of the no-intrusion version, which was used as a control. The recall means are shown in Figure 9*b*. Again there was a difference based on expertise ($F(1, 72) = 8{\cdot}05$, $p = 0{\cdot}006$). There was also a difference based on program version, with the beacon-like lines in the intrusion version being

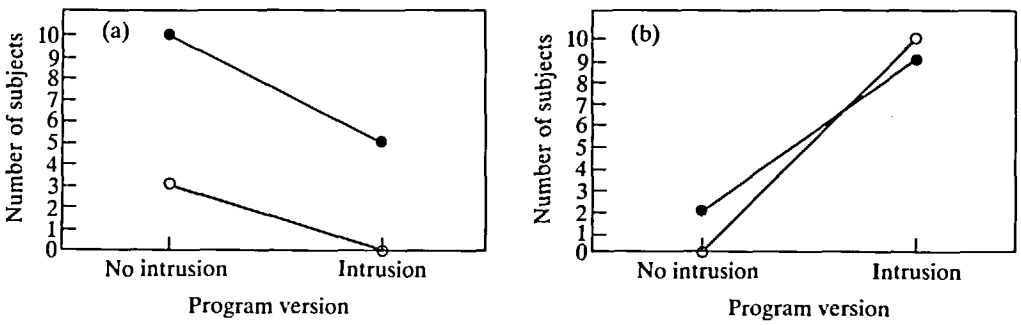

FIGURE 8. (*a*) Beacon intrusion experiment: number of subjects understanding program function as *binary search* ($N = 76$), (*b*) number of subjects understanding program function as *sort* ($N = 76$). ●, advanced; O, novice.

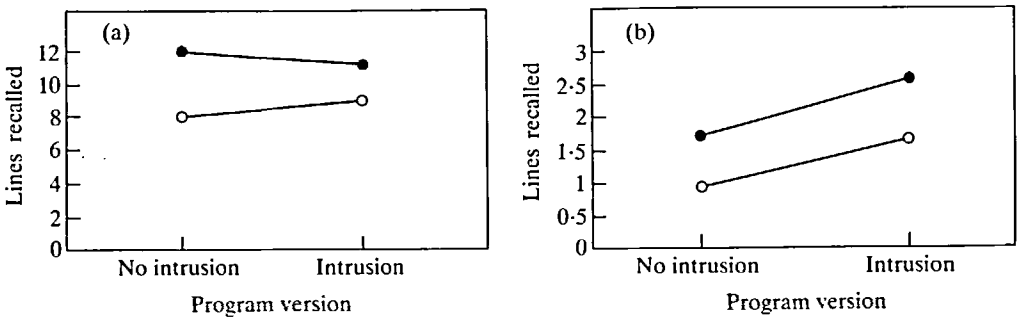

FIGURE 9. (*a*) Beacon intrusion experiment: mean recall of all 17 lines, (*b*) mean recall of lines 9–11. ●, advanced; ○, novice.

recalled significantly better than the correspondingly placed control lines in the no-intrusion version ($F(1, 72) = 8{\cdot}05$, $p = 0{\cdot}006$). It is notable that advanced subjects did very well at recalling the beacon in the intrusion version (a mean of 2·53 out of the 3 beacon lines). Yet looking at total recall of the programs they performed almost the same on the intrusion version, containing the beacon, as on the no-intrusion version, not containing the beacon. This may be some indication that advanced subjects had difficulty with the inconsistencies in the intrusion version. They certainly recognized the idiomatic beacon well but perhaps could not interpret its role in context, and this may have depressed their recall of the program as a whole.

5.5. DISCUSSION

The results of this experiment showed that the presence of a false beacon in a program has a strong tendency to mislead programmers about the program's function. Presented a simple program without an intruding beacon, many subjects were able to identify it's function correctly. However, presented the same program with a misleading beacon their ability to correctly identify the function declined strongly. Perhaps more importantly, the subjects who were misled about the function of the program containing the misplaced beacon interpreted it in accordance with the expectations set up by the beacon. This was true of both novice and advanced programmers.

In this experiment advanced subjects recalled the programs better than novice subjects. This could be explained simply by their greater experience of programming. Subjects recalled the beacon very well even when it was inappropriately placed in a program. Beacons are well-known idioms and tend to be very memorable to programmers even when the context they appear in is not wholly recognizable and coherent.

The picture which emerges from this experiment is that beacons are powerful enough indicators of function that they can lead programmers to false comprehension of a program, if they are inappropriately placed in it. Even though the program in which a beacon is inappropriately placed may contain other signs that conflict with interpreting it in accordance with the beacon, the expectations set up by the

presence of the beacon seem to dominate. Subjects tend to use the salient information from the beacon in making their initial judgements about program function, and they do not notice inconsistencies. Thus, the beacon appears to be a powerful guide in initial comprehension and would explain how such comprehension might go wrong. A task like debugging, requiring deeper processing, may be required to force subjects to go beyond the surface level.

6. Defective sort experiment

This experiment was designed to study further how beacons influence program comprehension. The first and second experiments reported in this paper suggested that a program which contains a beacon is more easily comprehended than one which accomplishes the same purpose but does not contain a beacon. However, it seems likely that other features of the program, in addition to the beacon, support the recognition of the program's function. For example, the sort programs which we have used in our studies contain a swap beacon that aids comprehension. In addition, however, a sort usually contains other characteristic features, such as nested loops and an array. These things may also be lower-order beacons which support and strengthen the interpretation of the primary beacon. In themselves, they are not strong indicators of function. (Loops or an array occur in many programs.) However, they form the environment in which the beacon is normally expected to occur. If they are missing, the programmer may be less likely to interpret the program in terms of the beacon. He or she may be more likely to search more deeply for other evidence about program function. There is some suggestion in the beacon intrusion experiment that subjects had some difficulty interpreting the role of the beacon in the intrusion version. In recall of the program, advanced subjects were very successful at recalling the idiomatic beacon but less successful at recalling the inappropriate surrounding context in which it was placed, perhaps indicating that they could not make sense of the program as a whole. The point of the present experiment was to see how beacons affect the comprehension of programs in the lack of other supporting features which would be expected to form the environment of the beacon. It was hypothesized that these supporting features are important to comprehension, and correct comprehension would decrease if they were absent, even when the beacon was present.

6.1. SUBJECTS

One hundred and sixty subjects participated in this experiment. Eighty were novices and 80 were advanced programmers. None of the subjects had participated in the other experiments. The subjects' backgrounds were similar to those in the previous experiment.

6.2. MATERIALS

The experimental materials consisted of four Pascal procedures. To test the hypothesis that the primary beacon has more influence on program recognition when expected supporting structures are also present, variations of a sort program were used. One of them was a correct and complete sort program. Others were

defective versions of the same program which were incorrect because they were missing various lines necessary to carry out the sort correctly. However, some of the versions were relatively close to the correct program and some relatively far from it.

The first experimental program was the correct sort procedure, referred to here as program CORRECT (Figure 10*a*). It was a standard Shellsort, containing 22 lines of code. The swap beacons was located at lines 12–14 ($t:=a[j]$; $a[j]:=a[j+k]$; $a[j+k]:=t$). This correct version contained the nested loops necessary for making repeated passes through the array. There were three loops, an outer While loop, a Repeat loop, and an inner While loop. The second version, referred to as program ONELOOP (Figure 10*b*), was incorrect and would not sort the array. However, it still was quite close to the correct program. It was 19 lines long and contained the swap beacon, identical to that in the correct program, at lines 10–12. The difference between this program and the correct version was that this one contained only the outer While loop. The Repeat loop and the inner While were deleted. Thus, the program bore a good deal of surface similarity to the correct sort, in that it had a typical looking swap and was contained in a loop. However, the nesting of loops was missing. Nesting of loops usually occurs in most elementary sorting procedures which employ swapping, so nesting might play a role as a secondary beacon. It was felt that the subjects would be less likely to say that ONELOOP was a sort based on the beacon because the nesting was missing. The third program version, program NOLOOP (Figure 10*c*) was based on the Shellsort procedure but was yet further from being a correct sorting program. The program was 17 lines long. It contained the same swap beacon used in the previous versions, located at lines 9–11. This version of the program contained no loops at all, but it retained all the other lines of the correct version. If the loops are necessary as a secondary beacon to support the recognition of the program as a sort, then this version should be much less likely to be called a sort. The effect should be stronger here than in the previous version where one loop was still present. The last program version used, program NOSWAP (Figure 10*d*), was a baseline for measuring the influence of the secondary beacons on comprehension, in the lack of the primary beacon. Here the Shellsort program was used, with its nested loops intact. However, the primary beacon thought to be most important for comprehension, the swap, was missing. The resulting program was 18 lines long. It was thought that this program was unlikely to be associated with sorting because of the missing swap lines, even though all the other lines of a correct sort were present.

Aside from the differences in content and length explained above, the programs were similar. All of them were derived from the same original correct Shellsort, so each defective program was a subset of the lines in the correct program. The variable names were the same in all cases. Each program was indented in the typical Pascal fashion. Because some programs contained less nesting of loops, some versions had fewer levels of indentation than others. Each program occupied about one-third to one-half of a printed page.

6.3. PROCEDURE

Twenty novice and 20 advanced subjects worked on each of the four program versions. No subject worked on more than one version. The experimental procedure

```
1  Procedure P7 (var a: atype; n: integer);
2  var i, j, k, t: integer;
3  begin
4    k:=n div 2;
5    while k>0 do begin
6      i:=k;
7      repeat
8        i:=i+1;
9        j:=i-k;
10       while j>0 do
11         if a[j]>a[j+k] then begin
12           if t:=a[j];
13           a[j]:=a[j+k];
14           a[j+k]:=t;
15           j:=j-k
16         end
17         else
18           j:=0;
19     until i=n;
20     k:=k div 2
21   end
22 end;
```

(a)

```
1  Procedure P8 (var a: atype; n: integer);
2  var i, j, k, t: integer;
3  begin
4    k:=n div 2;
5    while k>0 do begin
6      i:=k;
7      i:=i+1;
8      j:=i-k;
9      if a[j]>a[j+k] then begin
10       t:=a[j];
11       a[j]:=a[j+k];
12       a[j+k]:=t;
13       j:=j-k
14     end
15     else
16       j:=0;
17     k:=k div 2
18   end
19 end;
```

(b)

```
1  Procedure P9 (var a: atype; n: integer);
2  var i, j, k, t: integer;
3  begin
4    k:=n div 2;
5    i:=k;
6    i:=i+1;
7    j:=i-k;
8    if a[j]>a[j+k] then begin
9      t:=a[j];
10     a[j]:=a[j+k];
11     a[j+k]:=t;
12     j:=j-k
13   end
14   else
15     j:=0;
16   k:=k div 2
17 end;
```

(c)

```
1  Procedure P10 (var a: atype; n: integer);
2  var i, j, k: integer;
3  begin
4    k:=n div 2;
5    while k>0 do begin
6      i:=k;
7      repeat
8        i:=i+1;
9        j:=i-k;
10       while j>0 do
11         if a[j]>a[j+k] then
12           j:=j-k
13         else
14           j:=0;
15     until i=n;
16     k:=k div 2
17   end
18 end;
```

(d)

FIGURE 10. (*a*) Defective sort experiment: CORRECT program. (*b*) Defective sort experiment: ONELOOP program. (*c*) Defective sort experiment: NOLOOP program. (*d*) Defective sort experiment: NOSWAP program.

was the same as in the previous experiment except that no recall task was given because of the unequal length of the four program versions. In the experimental task where they described the function of the program subjects were told that if they felt the program was incorrect or accomplished no meaningful function they should say that and explain why in so far as possible. If they felt the program was incorrect

but they still had an idea of what it was probably intended to do, they were instructed to say what they thought the intention was.

6.4. RESULTS

The program function results are shown in Figure 11, which indicates how many subjects understood each program version as a sort. Three of the four versions were defective because they were missing lines and thus would not sort. The numbers listed in the figure represent only those subjects who called each version a sort *and* did not make any comment about it being incorrect or anomalous. In fact, very few subjects actually recognized a defect in the programs and said they were incorrect or meaningless. A $2 \times 2 \times 4$ Chi-square test was run on the program function data. One factor was level of expertise (novice/advanced) a second factor was whether the subject said the program was a sort, and the third factor was program version (four different versions as described previously). There was a significant difference based on program version ($\chi^2 = 36{\cdot}70$, d.f. $= 3$, $p < 0{\cdot}05$). Ryan's procedure for specific comparisons showed that only the NOSWAP version differed from the other versions, being much less likely to be identified as a sort ($p < 0{\cdot}05$). There was a significant difference based on expertise ($F(1, 152) = 5{\cdot}663$, $p = 0{\cdot}019$). Advanced subjects were more likely to say that the programs were sorts than novice subjects. The interaction between level of expertise and program version was not significant.

6.4. DISCUSSION

The experiment shows that subjects had a strong tendency to make use of a beacon in comprehending the program. This remained true even when other supporting structures, which are usually associated with the beacon, were absent. This result goes against the hypothesis of the experiment which was that programs would be less likely to be interpreted in terms of the beacon if other supporting structures were missing. It does, however, confirm the power of the beacon in directing comprehension. (For example, note that in the NOSWAP program, where the beacon is missing, only one subject called it a sort.) Apparently programmers are not strongly sensitive to the environment in which the beacon occurs on a quick first reading and study of the program. This was not particularly surprising in the ONELOOP program which still had its swap embedded in a While loop. However, it was more surprising in the NOLOOP program which simply had a swap with no looping mechanism at all. On deeper study, for example, looking for a bug, subjects

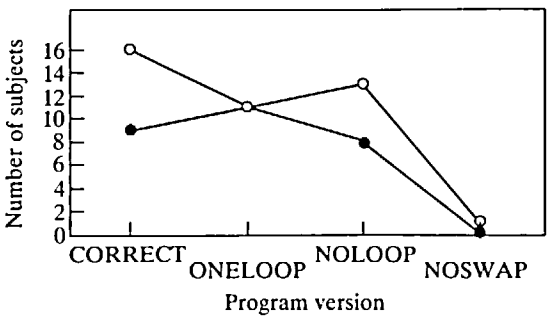

FIGURE 11. Defective sort experiment: number of subjects giving program function as *sort* ($N = 160$). ●, advanced; O, novice.

would certainly detect the discrepancy, but they did not do so in a quick initial study, where the beacon influenced their comprehension strongly. It should be noted that in this study many subjects tended to interpret the defective beacon-containing programs in accordance with the beacon, even though the directions suggested that there might be incorrect or meaningless programs in the experiment.

Advanced programmers relied on the beacon more heavily in comprehension than did the novices. This surely reflects the advanced programmers' stronger association, through experience, of the beacon with a particular program type. However, it had been expected that advanced programmers would be sensitive to the environment in which the beacon occurred. This was not the case. Advanced programmers were as likely as novices to say that the program versions lacking a plausible looping structure were sorts. Possibly, however, they would be quicker to detect the defects in the programs if given a task requiring deeper analysis of the code.

This experiment indicates that beacons have considerable power in molding initial program comprehension. Even when a program is incorrect and deviates fairly strongly from what could conceivably be a correct program, the presence of the beacon leads programmers to interpret the program in accordance with the beacon. They fail to notice large discrepancies in the surrounding code, or, if they do notice, they still fail to look for alternative interpretations. On initial study they assume that the program must carry out a function consistent with the beacon. There was no strong evidence for the importance of secondary, or supporting, beacons for the overall, high-level comprehension task performed here.

7. Conclusions

Some of the experiments reported in this paper contained design difficulties, as explained previously. However, taken together we believe that the results converge to provide an accumulation of evidence about the role of the beacon studied. The beacon disguise and prototypical and non-prototypical sort experiments support a connection between the presence of the beacon and program comprehension. This connection seemed stronger for more experienced programmers, as shown by the interactions. The meaning of these findings is that the idiomatic, or stereotypical, code did appear to play a large role in initial high-level comprehension of programs. It helped the programmer to determine overall program function quickly, with minimal effort, and based on scanty information. While the time limitations which subjects worked under here were artificially imposed by the experiment, programmers may impose on themselves some limits of time or effort in the initial stages of program scanning and comprehension. Such self-imposed limits would make sense if the purpose is to gain a general familiarity and mental map of the program before proceeding with some task that requires detailed study and simulation. Looked at in this context, the beacon was comparable to a topic sentence or perhaps, better yet, to keywords in prose. It permitted quick, efficient high-level comprehension without line-by-line study of the text.

The recognition of stereotypical code comes from programming experience. Likewise the connection of some piece of stereotypical code with a certain program function comes from repeatedly seeing that same idiom used to accomplish the given function, though perhaps in slightly different variations. Thus, both the beacon itself and its usual association with one operation are learned. Because all of this

depends on experience, it must develop gradually. In our experiments we expected to see the use of the beacon in more experienced programmer and, indeed, this was sometimes true. However, in some cases the novices seemed almost as adept as the advanced subjects at utilizing the beacon employed in these studies. Clearly, the ability to recognize this simple and clear beacon begins to develop early.

Some other possible characteristics of beacons also emerge from these studies. The intrusion experiment showed that a strong beacon was a powerful enough pointer to program function that is led to false comprehension. Apparently, on a quick orienting study of a program, programmers simply looked for a familiar entity and based judgements on it. There was no real effort to find evidence to support an initial judgement about program function. Perhaps at this level of study discrepancies between a beacon and other surrounding contextual information are not noticed. This seems particularly likely when, as in the case studied here, the beacon in question is very strongly and uniquely associated with some particular structure or operation and is unlikely to occur in other contexts. This tendency to interpret code in accordance with a strong beacon is an example of how expertise can sometimes hurt performance.

Brooks (1983) suggests that programs or procedures may contain multiple beacons, and he introduces the idea of the diagnosticity of beacons. Some beacons may be strongly diagnostic and lead to strong and definite conclusions about program function. Other beacons are not so diagnostic by themselves and may require other supporting beacons to be present before a conclusion can be made about function. Even for a strong, highly diagnostic beacon, other features which typically co-occur with it may have to be present for a programmer to draw a conclusion. These studies did not explore fully the question of beacon diagnosticity and the co-occurrence of beacons in programs. The beacon that we were working with was chosen because it appeared to be quite diagnostic. However, the defective sort experiment did center on the environment of the beacon and the support which the environment gives to judgements about function. It appeared that in the initial study of the programs the strongly diagnostic swap beacon was relied on heavily for comprehension. Other features which typically occur in combination with it were less likely to be noticed. The programmers failed to detect contradictory signs present in the program. Thus, in early comprehension the beacon was much more salient than the overall plan of which it was a part. For a weaker beacon, not so uniquely associated with a given structure or operation, this tendency to overlook supporting information may be less pronounced, although the present experiments did not address that.

In this paper we have described beacons as idioms in programming or stereotypical code. However, there are many idioms in programming which are not beacons. The prototypical and non-prototypical sort experiment contained an example of this in an initialization loop which was well-known and highly memorable, but knowledge of which did not correlate well with program comprehension. Some idiomatic code may simply be too widely used in too many circumstances to be very diagnostic for a given program. However, such idioms might still be beacons of secondary importance which become important when the primary beacon is not completely diagnostic or when a deeper study of the program is carried out.

The experiment reported here showed that the swap serves as a beacon in sort programs. If it is absent or wrongly placed in some other program, comprehension

deteriorates. A limitation of this research is that we concentrated on one program type and one beacon. This was done because they seemed to be ideal examples that would give the best chances of finding an effect. Now the problem is to find out whether beacons are of wider importance in different types of programs. This is necessary in order to be able to generalize about beacons as a class. Therefore, the results here are suggestive but need to be expanded. To achieve this expansion, we first need to find an independent way, other than experimenter intuition, of identifying beacon-like structures in program code. Then we can test the role of other beacons and compare to results found here. Work on finding an independent method to identify beacons is now underway. In addition to generalizing to beacons as a class, we would also like to study their role in realistic sized programs. We would like to find out if large programs written to take advantage of stereotypical code (such as programs produced by the Programmer's Apprentice) are understood more quickly on initial study and whether this has an ultimate influence on the efficiency of accomplishing other programming tasks. A study of large programs is necessary to determine whether beacons play a role in real comprehension tasks.

This research was supported in part by Grant 8808424 from the National Science Foundation.

References

BROOKS, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies*, **18**, 543–554.

EHRLICH, K. & SOLOWAY, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. C. THOMAS & M. L. SCHNEIDER, Eds. *Human Factors in Computer Systems*. Norwood, NJ: Ablex.

JEFFRIES, R. (1982). A comparison of the debugging behavior of expert and novice programmers. Paper presented at *The American Educational Research Association Annual Meeting*.

JOHNSON, W. L. & SOLOWAY, E. (1985). PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering*, **11**, 267–275.

KANT, E. (1985). Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, **11**, 1361–1374.

KNUTH, D. E. (1973). *Sorting and Searching*. Reading, MA: Addison-Wesley.

LITTMAN, D. C., PINTO, J., LETOVSKY, S. & SOLOWAY, E. (1986). Mental models and software maintenance. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

RICH, C. & WATERS, R. C. (1988). The Programmer's Apprentice: a research overview. *Computer*, **21** (11), 10–25.

RIST, R. S. (1986). Plans in programming: definition, demonstration, and development. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

SOLOWAY, E., EHRLICH, K., BONAR, J. & GREENSPAN, J. (1982). What do novices know about programming? In A. BADRE & B. SHNEIDERMAN, Eds. *Directions in Human/Computer Interaction*. Norwood, NJ: Ablex.

SOLOWAY, E. & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **10**, 595–609.

WIEDENBECK, S. (1986*a*). Processes in computer program comprehension. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

WIEDENBECK, S. (1986*b*). Beacons in computer program comprehension. *International Journal of Man–Machine Studies*, **25**, 697–709.