MAKING PROGRAMS MORE READABLE

J. L. Weiner(1) and R. M. Burstall(2)(3)

Abstract: In this paper we make suggestions aimed at enhancing the readability of programs. Our suggestions are about structure - the underlying structure of a program, and the structure of its text. We show how the types of relations expressed by the underlying structure of a program can be extended, and how these extended relations can be conveyed by its text, so as to enhance the readability, and ultimately the understandability of a program. We also show how the readability of the program can be improved by having its text linearly ordered so that each sentence motivates upcoming ones. This will complement the traditional notion of sequential program control flow.

---

(1) Dept. of Mathematics and Computer Science, University of New Hampshire, Durham, NH, USA

(2) Department of Computer Science, University of Edinburgh, Edinburgh, SCOTLAND

(3) This work was conducted while both authors were members of the Department of Artificial Intelligence, University of Edinburgh, Edinburgh, SCOTLAND.

1. INTRODUCTION

A common phenomenon among programmers is to rewrite programs others have written, because they can't understand exactly what it is the programs do, or why they were written the way they were. It is also not uncommon for programmers to rewrite their own programs, because after a while they too can't understand them. Often rewriting a program involves a cycle; starting off by a total rewrite, and eventually coming back to the original program, because the programmer has rediscovered what he or the other programmers eventually understood when the program was originally written.

Proposed solutions to this problem usually include using abstractions, modules, and comments. However, there are problems with each of these partial solutions; not in their intent, but in the realization of them in the program's text. In this paper, we will make several suggestions aimed at enhancing the readability of programs.

When we say we want to enhance the readability of a program, we are really saying that we want to enhance the likelihood that a person can understand a program simply by reading it, rather than having to rely on some external explanatory text.[4]

Our suggestions will be about structure - both the structure of the program's text, and its underlying (semantic) structure. Since programs are read line by line (sequentially) the linear ordering of their text is extremely important. Basically, we believe that text should be ordered so as to provide the motivation necessary to understand what is to come next. It is important to stress the word "necessary", since information given too soon will not motivate, but only add to the complexity and unreadability of the program. We will be mainly interested in the ordering of declarative statements, although some suggestions will be made about the order of imperative statements as well.

Traditionally, the hierarchical organization of a program has been considered to be its underlying structure. However, we believe that this describes only one way in which parts of a program can be related, and we will suggest others, such as qualification.

We will also suggest better ways in which the text of a program can convey its underlying structure, since understanding a program requires understanding its underlying structure.

It should be stressed that the constructs we will introduce to make programs more readable have a systematic basis in natural language as observed experimental-

---

[4] Of course, a program is not written in isolation, but within a certain context, and it must be assumed the reader is familiar with that context.

ly. They were, in fact, suggested as a result of a careful analysis of people's descriptions of their programs (see [Weiner and Burstall, 1979] for details). These descriptions were taped and analyzed, using the methodology of discourse analysis [Linde, 1979] to determine the overall structure of the discourse.

The use of discourse analysis in the design of programming languages is motivated by two observations. The first is that little is known about the way programs should be structured (see [Wegner, 1979] for a discussion). This is in contrast to the amount of knowledge that people have about how a conversation should be structured [Labov, 1972] [Linde and Labov, 1975] [Linde and Goguen, 1978]. We believe therefore, that much can be learned from the way people structure language and applied to the design of programming languages.

The second observation is that the structure of a programming language must be consistent with the structure of the language people use to describe programs. If it is not, it will be hard to learn, due to its unnaturalness. It will also be hard to use, because it will require a person to have two different structures in mind when programming. One structure being the way the person naturally structures a program, and the other a translation of that into the structure of the programming language used.

The idea of studying the discourse structure of program descriptions (which led to the present work) was suggested to us by Joseph A. Goguen, and came out of his joint work with Charlotte Linde which has been the main influence on us.

2. REFERENCE

One of the first things programmers learn is to give variables meaningful names. A meaningful name is usually one which indicates the function that the named object plays. In this way, it is possible to associate some semantics with otherwise indistinguishable structural objects. This enhances the readability of the program and allows the programmer to desk check it for logical errors, such as comparing apples to oranges.

However, the semantics are only in the programmer's mind. To associate some semantics with an object in the program requires defining it as a new data type.[5] Thus a meaningful variable name is really the name of a data type. For if an object has a meaningful name, it can be distinguished from others by its semantics, and thus ought to be by a compiler.

---

[5] It has been suggested by D. MacQueen that the term "type" is inappropriate here, since the semantic properties and the type exist independently of each other. Instead, one should say: x is of type t with properties p. Although we agree, since we want to talk about something with both a form and semantics, rather than coin a new term, we will continue to naively use the term "type".

It still might be argued that one should distinguish between types and instances of them, which still need to be referred to by a variable name. For example, "man" the type, and "mailman" the instance of the type. But this is just confusing the relation of an instance with a more important relation of subtype. A mailman might for some purposes be best considered as man enriched with additional properties.

Not all objects can be distinguished uniquely structurally or semantically, and therefore we need another way to refer to these objects. Rather than use meaningless variable names, such as tree1 and tree2, we will refer to them as the first tree, second tree, etc. in a n-tuple of trees.

It could also be argued, as Winograd does in [Winograd, 1979], that it is impossible in current programming languages to define a sufficiently rich semantics for objects and relations between them. This is undoubtably true, although languages such as CLEAR [Burstall and Goguen, 1977] and ZENO [Ball et al., 1978] are important steps in the right direction. Despite this drawback, we will abandon the use of variables in this paper.

Our decision to abandon variables is reinforced by observations of how people refer to objects, which is again, by their function. The function an object performs depends on a person's perspective on it, and may change as the perspective changes (see [Bobrow and Winograd, 1977])[6]. For example, in an ELIZA program [Weizenbaum, 1965], a sentence is viewed as a set of words when transforming it from the first to the third person, and then as a set of phrases when trying to match it against key phrases stored in its dictionary.

We will also adopt the convention that people have of qualifying a type name with the name of an action that it was applied to, so as to indicate a subtype. For example, a sentence which has been transformed is termed a "transformed sentence". When such a subtype appears as the argument to another procedure, it clearly indicates the dependency of one action on another. For example, in ELIZA, part of the process of building a sentence to output involves searching a sentence for key phrases. Since one of the phrases is "you", this action is dependent on the sentence having been translated from first to third person.

3. TIEING IT ALL TOGETHER

A book has as its focus a particular topic, be it a person, place, event, etc., and it is organized around that topic. The topic is usually introduced at the

---

[6] A similar observation was made by K. Ehrlich [Sussex] while performing an experiment in which people were asked to build some large object from smaller ones. For example, when building a car, one piece was referred to as a mudguard, and when building a house, the same piece was referred to as a roof.

beginning of the book, and then the rest of the book is about it. Other topics may be introduced later in the book, but they are assumed to be related to the original topic. If not, the book is considered to be incoherent. This is true of most other types of text as well. A program being a notable exception; there is no construct available in a programming language to introduce a topic. Programs tend to be unreadable because there are no explicit topics to tie everything together. Even if there is an implicit topic, the organization of the program obscures any recognition of something being a topic. In human discourse there are features of language, such as anaphora, which are used to reinforce the fact that something is a topic or subtopic throughout the text. In this section we will suggest a programming language construct to introduce a topic, and suggest other changes that are implied by the use of this construct.

In terms of a programming language, we will consider a topic to be represented by a data type. To introduce a topic we write:

Take a <data type>

as in:

```
Take a grade
        grade := grade + 50
        if grade > 90 then grade := grade / 2
                      else grade := grade / 1.5
```

a program to scale a grade.

Some programming languages allow the above program to be shortened into:

```
Take a grade
        grade +:= 50
        if grade > 90 then grade /:= 2
                      else grade /:= 1.5
```

since the identifier on the right-hand side is redundant in this special, but frequently used case.

In English, after a topic has been introduced, we are able to refer to it anaphorically. For example, with pronouns such as "it". (Some programming languages [Gordan et al., 1979, Papert, 1972] also allow limited use of the anaphore "it".) We can rewrite the program, using "it" as follows:

```
Take a grade
        it +:= 50
        if it > 90 then it /:= 2
                   else it /:= 1.5
```

This program is reasonably easy to read if we make several conventions: "x +:= y" is read as "add y to x", "x > y" is read as "x is greater than y", and "x /:= y" is read as "divide x by y". When speaking, it is assumed that the result of either the addition, or division operator will be subsequently referred to by "it". That is, "it" has been side-effected. However, this doesn't mean that the original grade no longer exists, has in some way been overwritten, it simply means that the pronoun "it" refers to the current grade.

Indicating all the conventions necessary to read a program is cumbersome. Rather than do this, we will adopt a distributed-fix notation, as is used in OBJ [Goguen and Tardo, 1979]. For example, to define "+:=" we write add_to_. The underbar indicates a parameter to the function. The same program written in distributed-fix notation is given below:

```
Take a grade
        add 50 to it
        if it is greater than 90 then divide it by 2
                                 else divide it by 1.5
```

In any but the most trivial programs, we will want to refer to more than one object. To do this we embed introductions, as in:

```
Take a grade
        Take a scale
                add it to the grade
```

To refer to anything other than the innermost topic, we use a definite referent.

If the topic introduced is a structured object, we can refer to its parts without qualifying them with the topic. Thus an introducing a topic has the same effect as using the *with* construct in PASCAL [Jensen and Wirth, 1974]. For example, suppose we have an object PAIR defined as:

```
type pair = record
            first_element:integer
            second_element:integer
            end
```

we can then write the following program:

```
Take a pair
        print the first_element
```

which will print the first element of the pair. Thus introducing topics allows us to use ellipsis.

The actual rules for determining what is in scope, how topics are referenced when in scope, and how possible ambiguities will be handled is quite complex, and will not be further elaborated upon here.

The choice of a topic indicates its importance; it also suggests that other topics are of lesser importance. In this way, we will create a hierarchy of topics This is similar to the situation of actors in a movie, some play leading roles and others supporting roles. This should not be confused with the hierarchy created by nesting blocks in a block-structured language. Although it is related, it is not the same. All variables in a block are declared at the front of the block, regardless of their importance. Therefore the hierarchy is not consistent.

Another difference is that efficiency is not a consideration when structuring a program into topics, but it is when structuring a program into blocks. In fact, there is no relationship between the introduction of a topic and allocation of storage for it.

It should be obvious from the discussion above, that introductions can be nested, and that the environment created by a nested introduction automatically inherits the topics of the introductions in which it is embedded. In effect, this allows globals. This scope rule should be contrasted with other recent proposals, such as in EUCLID [Lampson et al., 1971]. In these proposals, the scope of a variable is not automatically extended as it would be in ours, rather an extension must be requested. These proposals are in response to criticisms of globals as inducing errors because of their scope.

We believe that globals can be handled successfully by our proposal, because we maintain a consistent hierarchy of topics which is reflected in the underlying structure of the program. That is, it is part of a person's understanding of the program. We view actions as being subordinate to objects, rather than vice versa. In this view, the relation of an object to an action is that the action is *about* the object, not that an action uses an object which is the usual relation in a programming language. The fact that actions are about an object reinforces the importance of an object as a topic. In addition, the fact that any subtopics introduced are only relevant by their relation to the topic, also reinforces this view. As a result of this view, a topic should always be attended to, not forgotten, and thus not induce errors. Of course, an unlimited nesting of topics would be confusing, and we assume that people will naturally restrict them as they do in conversation, so as to be understood.

Since an introduction only focuses attention on an object, and does not create it, not all objects are subject to being introduced, only those that exist prior to their introduction. Some objects are created as the result of some actions, and are therefore not explicitly introduced. For example, in the following program the total grade is not explicitly introduced:

Take a grade
add *it* to *the* total grade

To introduce an object prior to its creation would only add to the complexity and unreadability of the program, because at that time, knowledge of it is not necessary for an understanding of the program. A text should be organized so that new information is not given all at once, but rather incrementally.

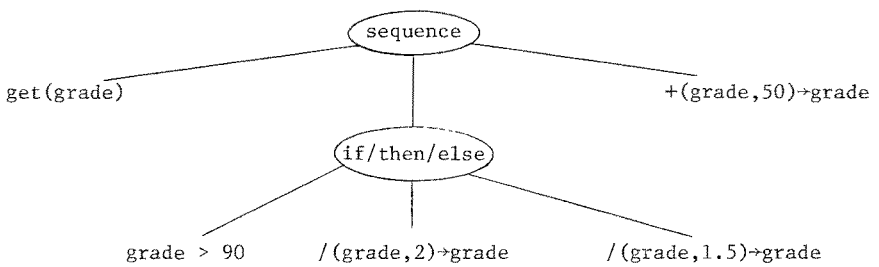
There is an error in the above program. The total grade should have been initialized, but hasn't. Since initialization is a relatively unimportant action, its status as such should be conveyed in both the underlying structure of the program, and the text. To convey its status we must subordinate the initialization to the introduction (implicit or explicit) of the object. This can be done by using the *where* construct, as in:

add *it* to *the* total grade
where the total grade is initially zero.

One problem remains. We cannot, in general, expect current compilers to check that the use of topics in a program is coherent. We must rely on people's judgement. At least, the use of a topic provides an additional criterion on which to judge a program's composition.

4. UNDERLYING STRUCTURE AND PROGRAM TEXT

Suppose we have a tree representing the underlying structure of a program written in some abstract language, as in:



where each level in the tree represents a different level of abstraction.

We can translate the program represented by the tree into PASCAL:

```
var grade:integer;
begin
        read(grade);
        if grade > 90 then grade := grade / 2
                      else grade := grade / 1.5;
        grade := grade + 50;
end
```

and into LISP

```
(LAMBDA (GRADE)
        (COND ((> GRADE 90) (SETQ GRADE (/ GRADE 2)))
              (T (SETQ GRADE (/ GRADE 1.5))))
        (SETQ GRADE (+ GRADE 50))))
```

If these programs are to be understood, their underlying structure must be conveyed in the text, or as Knuth points out in [Knuth, 1974], "... a good program will be composed in such a way that each semantic level of abstraction has a reasonably simple relation to its constituent parts".

Before we see how these programs convey information about their underlying structure, some terminology will be introduced. Think of a parser building up a parse tree. At some point during the parse, a partially built tree exists and there is a pointer to the node where the next node will be added. The operation which moves the pointer down and adds a new level onto the tree will be called a PUSH. Conversely, we will call POP the operation which moves the pointer up the tree to a previously added node when the current level is terminated. A syntactic

marker which indicates when a PUSH or POP operation should be invoked is called a PUSH and POP marker respectively.

In the PASCAL program, *begin* and *if* are PUSH markers, and *end* is a POP marker. There is no explicit POP marker corresponding to *if* in PASCAL, although in other languages, such as ALGOL68, there are. In LISP, the PUSH marker is a left parenthesis, and the POP marker is a right parenthesis. In some LISP systems, a right bracket indicates that several levels are to be popped.

Although these markers are adequate for a parser to detect the appropriate movement in the tree, they are inadequate for people. To convey information about the underlying structure of such programs to people requires, at least, that different levels be indicated by different amounts of indentation.

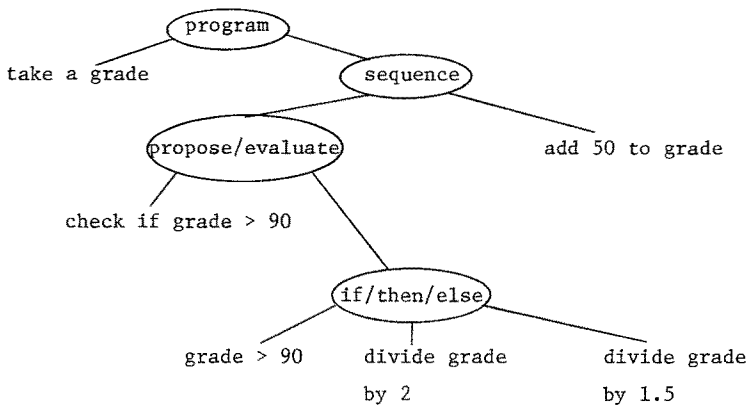
Many types of human discourse have a structure which can be expressed as a tree, and in which POP and PUSH markers exist. We believe that these markers used in human discourse should be incorporated into programming languages, rather than, as is now done, artificial ones invented. The futility of using artificial markers is demonstrated by the incomprehensibility of programs, as compared to the understandability of human discourse. As Winograd observes [1979]:

> We cannot turn programmers into native speakers of abstract mathematics, but we can turn our programming languages in the direction of natural descriptive forms.

Indentation serves not only to indicate the underlying structure of a program, it also serves as a form of paragraphing. This allows a programmer to organize a program so that its parts can be detected visually. As a form of paragraphing, indentation is somewhat restrictive, so instead, we will use a more general form. Since our observations are about oral discourse, we will make no claims about the adequacy of the particular form of paragraphing we have chosen to use in this paper

5. ADDITIONAL CONSTRUCTS AND THEIR ROLE AS SYNTACTIC MARKERS

In this section we will suggest additional language constructs and indicate their role as POP and PUSH markers. As with the previous section, we will precede by first specifying the underlying structure of a program:



The above structure corresponds to the following program:

Take a grade and *check if it* is greater than 90

If it is *then* divide *it* by 2 *otherwise* divide *it* by 1.5

Now add 50 to *it*

The first thing to note is that the *take* construct is a PUSH marker. It indicates that the rest of the program (or until the topic changes) will be about it. *Take* also has interesting semantics. In this case, since grade is referred to by an indefinite referent, and has not been initialized, it indicates that an instance of the grade data type should be created, and its value read in from the terminal. If grade had been referred to as "the grade" (a definite referent), the current environment would be searched for the grade, and that grade would be placed into focus.

Still another case exists, illustrated by the following program:

Take a set of grades

Take a grade *from that* set, and add *it* to *the* total grade.

Repeat for all other grades

the first introduction indicates that the operation accepts a set of grades as a parameter. The second introduction refers to an object from the aggregate[7] introduced on the line above. If the aggregate is unordered, as is a set, then the object referred to by the nested introduction is a random object taken from the aggregate. If the aggregate is ordered, as is an array, then the nested introduction refers to the first element.

---

[7] The term "aggregate" is used here to refer to a set or sequence of a uniform element type.

This program contains a loop. Each time through the loop "it" refers to another grade in the set of grades until all the grades have been taken. The loop begins with the introduction of an object from an aggregate, and terminates with the *repeat* construct, which indicates that the loop should be repeated. The actual *repeat* construct has an optional *until* clause which specifies an additional condition on which to terminate the loop.

The structure of this loop differs from that of most others. Firstly, the beginning of the loop is not explicitly marked. It is only implicitly marked by the semantics of the introduction. Why would an indefinite object from an aggregate be introduced, except as a prelude to some form of iteration? Since the semantics are enough, there is no need to clutter the language with an additional construct. Secondly, a specification of how the aggregate is iterated over is not given until the end of the loop. By delaying this specification, the actual method of iteration should be clearly motivated by the text of the loop, and thus its choice easily understood. Another reason is that it serves a dual purpose. It signals the end of the text (and thus is a POP marker), and also indicates that the loop is to be repeated. A loop with the syntax "repeat x until y" is confusing, since it seems to say that the text (x) should be repeated, not its execution.[8] The *check* construct introduces a sequence of boolean conditions. As an introduction, it both motivates what is to come next by enumerating the conditions, and signals a PUSH. The term "otherwise" is used, because it is more natural than "else". *Otherwise* is both a POP and a PUSH marker, since it signals the end of the *then* part, and the beginning of the *else* part. If the *then* part is more complex than a single statement, another *if/then* statement must be used in place of the *otherwise*, since it reintroduces the boolean condition which might have been forgotten while reading the *then* part. This does not mean that the condition is reevaluated though. Only one *then* part is ever expected, and in this way the semantics are similar to *cond* in LISP.

The only other new construct illustrated by the program is *now*, which signals a POP of one level.

6. JUSTIFICATIONS AND COMMENTS

Wulf argues in [Wulf, 1979] that "... the various abstraction and refinement steps [in a top down design] are not necessarily present in the final program. Thus the methodology [of top-down design] serves well during the initial development, but fails to help the future program modifier".

We believe that Wulf's argument does not point out problems with top-down design, but with programming language design. That is, there is no convenient way to

---

[8] In this way it is similar to a loop proposed by Ole-Johan Dahl [Knuth, 1974]. This loop construct was, in fact, one of the surprising results of our analysis.

add information about the abstraction and refinement steps of a program's development into the final text. One might think that this could be done by adding comments, but comments are not part of a programming language. Programming languages might allow arbitrary text to be interspersed with program text, but they are still distinct. There is also no way to structure comments, because they are not part of the underlying structure of the program.[9]

If we intersperse program text with comments, we can easily see just how distinct they are:

```
{compute the average grade of the ith course by}
begin
     total_grade := 0;
     number_of_grades := number_of_students;
          {which is the maximum number possible}
     {now calculate the total grade by}
     for j:=1 to number_of_students do
     begin
          gr := grade [i,j]
               {this array contains the grade for the jth student in the
                ith course}
          if  gr /= 0
               then {the student took the course so}
                         total_grade := total_grade + gr
               else {the student didn't take the course so}
                         number_of_grades :=
                                   number_of_grades - 1;
     end
     {now calculate the average grade by}
     if number_of_grades /= 0
          then average_grade :=
               total_grade / number_of_grades;
          else average_grade := 0;
end;
```

adopted from Alagic and Arbib [1978].

We suggest that information about the development of a program can be added into the underlying structure by extending the types of relations expressed by tha structure. Furthermore, we suggest that the relations added be *comment-on* and *support*. These relations relate code to comment, and code to justifications respectively. Of course, adding developmental information to the structure is not enough, we must also add it to the language. Since the structure expresses more than one relation, we must signal the appropriate relation in the text by an appropriate syntax. We illustrate the syntax in the following program[10]:

---

[9] Although some LISP systems and MENTOR have the ability to prevent "comment migration" when pretty printing, comments are still not part of these languages, because they provide no notion of how comments should be structured.

[10] Assume that the appropriate data structures have been defined which relate an average grade to a set of students.

Take a course *and calculate the* average grade in *it by doing the following:*

Take a student *from the* set of students *and take her* grade. *Check if it is* zero.

If it is, *then* the student didn't take the course so subtract one from *the* number of grades *where that* number was initially equal to *the* number of students *which* in turn represents the maximum number of grades possible for a course.

Now if it is *not* zero *then* the student did take the course *so* add *it* to *the* total grade *where that* was initially zero, *and repeat for all other* students.

Now calculate the average grade *by doing the following:* Divide *the* total grade by *the* number of grades *except if the* number of grades is zero. *If so, then the* average grade is zero.

As we can see from this program, comments are signalled by "which", and justifications by the sentential connectives "because" and "so". One type of abstraction is signalled using "by" which relates the specification of the goal with the method for achieving it. Another example is given in the appendix.

7. CONCLUSIONS

In summary, the work described in this paper involved two activities:

1. Trying to discover what are the natural narrative devices for expressing algorithms.
2. Trying to devise semiformal constructs to support these devices.

A language is currently being developed at the University of New Hampshire which incorporates the constructs described in this paper. Also being developed is a system to translate programs written in this language into PASCAL. It is hoped that with this system experiments can be performed to evaluate the ability of these constructs to enhance the readability of programs.

ACKNOWLEDGEMENTS

We are indebted to Joseph A. Goguen for suggesting the direction pursued in this paper, and for, in collaboration with C. Linde, providing the foundation which made this work possible. We would like to thank M. Gordon, D. MacQueen, C. Mellish, R.L. Schwartz, and J. Scott for their helpful suggestions and criticisms, and Annette Aldrich of the Word Processing Center for preparation of this paper. We would also like to acknowledge the support of the Science Research Council.

BIBLIOGRAPHY

Alagic, S. and M. A. Arbib, The design of well structured and correct programs, Springer Verlag, New York, 1978.

Ball, J., Williams, G., and Low, J., "Preliminary ZENO language description", Technical Report TR41, Computer Science Dept., University of Rochester, December, 1978.

Bobrow, D.S. and Winograd, T., "An overview of KRL, a Knowledge Representation Language, Cognitive Science 1(1), March, 1977.

Burstall, R.M. and J. A. Goguen, "Putting theories together to make specifications", in IJCAI, MIT, 1977.

Goguen, J.A. and J.J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", in Proceedings IEEE Conference on Specification for Reliable Software, pages 170-189. IEEE, 1979.

Gordon, M., R. Milner, and C. Wadsworth, "Edinburgh LCF: A Mechanized Logic of Computation", Technical Report, Department of Computer Science, University of Edinburgh, 1979.

Jensen, K. and N. Wirth, Pascal Users Manual and Report, Springer Verlag, Berlin, 1974.

Knuth, D.E., "Structured programs with goto statements", Computer Surveys 6(4), December, 1974, ed. P. Denning.

Labov, W., "The transformation of experience into narrative syntax", in Language in the Inner City, University of Pennsylvania Press, Philadelphia, 1972.

Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, "Report on the programming language EUCLID", SIGPLAN notices 12(2), February, 1971.

Linde, C. and J.A. Goguen, "Structure of Planning Discourse", J. Social Biol. Struct. 1, 1978.

Linde, C. and W. Labov, "Spacial Networks as a site for the study of language", Language 51, 1975.

Linde, C., "The organization of discourse", in The English Language: English in its Social and Historical Context, Winthrop, Boston, 1979, edited by T. Shopen, A. Swicky, and P. Griffen.

Papert, S.A., "Teaching children thinking", Programmed Learning and Educational Technology 19(5), September, 1972.

Wegner, P. (editor), Research Directions in Software Technology, MIT Press, Cambridge, 1979.

Weiner, J.L. and R.M. Burstall, "Some observations about program descriptions", 1979, in preparation.

Weizenbaum, J., "ELIZA - A computer program for the study of natural language comprehension between man and machine", Communications of the ACM 9, 1965.

Winograd, T., "Beyond Programming Languages", CACM 22(7), July, 1979.

Wulf, W.A., "The next generation of programming languages, in Perspectives in Computer Science, ed. A.K. Janes, Academic Press, New York, 1977.

APPENDIX

EIGHT-QUEENS PROBLEM

In this appendix, we illustrate a solution to the eight-queens problem:

Take a chessboard *where* every space is empty *and take a* set of 8 queens.

Find a configuration for *those* queens on *the* chessboard *which* has the property that no queen may be taken by any other queen. *Accomplish this by doing the following:*

Take a queen *and find a* safe square on *the* first empty row to place *it* on, *where a* safe queen is *one* on *an* empty column, *and* on *an* empty up and down diagonal. *Now accomplish this by doing the following:*

Take a square on *that* row, *check if it* is safe.

If it is, *then* place *the* queen on *that* square, and *check if the* board is full *by doing the following:*

Check if this queen is *the* last queen.

If it is, *then* the board is full *so take a* queen, print *it* out *and repeat for all other* queens.

Now if the square is *not* safe, *then repeat for all other squares until a* safe *one* has been found.

Now if a safe one has *not* been found, *then* we have to backtrack *so take the* previous row *and* remove *the* queen from *it and find another* square on *that* row to place *the queen* on.

Except if that square is *the* last square, *then* there is no chance to find a safe square on this row *so repeat with another* previous row.

If a safe *one* has been found, *then repeat for all other* queens.