N73-19213

SYNTAX-DIRECTED DOCUMENTATION FOR PL360*

Dr. Harlan D. Mills
IBM

PL360, due to the efforts of Niklaus Wirth (ref. 1), is a phrase-structured programming language which provides the facilities of a symbolic machine language for the IBM 360 computers. It is defined by a recursive syntax and is implemented by a syntax-directed compiler consisting of a precedence syntax analyzer and a set of interpretation rules, as discussed by Wirth and Weber in reference 2.

Syntax-directed documentation refers to an automatic process which acquires programming documentation through the syntactical analysis of a program, followed by the interrogation of the originating programmer. This documentation can be dispensed through reports or file query replies when other programmers later need to know the program structure and its details.

The interrogation of an originating programmer consists of a relisting of the program text, with certain syntactic entities, which are classified as documentation units, set off typographically in lines and labeled with an ordinal coordinate system and a sequence of questions about these documentation units. These questions are generated automatically by completing prestored skeleton questions with coordinates and/or programmer-generated identifiers. The programmer's responses to the questions are stored and indexed to these documentation units for retrieval.

A key principle in what follows is that the programming documentation process is managed solely on the basis of the syntax of programs. The semantics of the documentation, as embodied in programmer responses to interrogation, are not analyzed by the process except in mechanical ways such as keyword indexing. In this way, a programmer's responses are treated as "black messages" in the process, in analogy to the idea of a "black box." That is, a programmer's responses are requested, accepted, stored, and later retrieved with no semantic analysis of their contents.

SYNTACTIC PRELIMINARIES

We use the notation and definitions for PL360 in reference 1. In defining documentation units and lines, the following device is used. First, denote the grammar in reference 1 by G, which defines the language PL360, $L(G)$. This grammar G will be transformed finitely into a new grammar G^* such that

$$L(G^*) = L(G)$$

```
 1   <BLOCK>              ::= <BLOCKBODY> END
 2   <CASE ST>            ::= <CASE SEQ> END
 3   <FOR ST>             ::= FOR <ASS STEP> <LIMIT> <DO> <STATEMENT*>
 4   <FUNC DECL7>
 5   <FUNC ID>
 6   <FUNC ST>            ::= <FUNC1> )
 7   <GOTO ST>            ::= GOTO <ID>
 8   <IF THEN ELSE ST>    ::= IF <COND THEN> <TRUE PART> <STATEMENT*>
 9   <IF THEN ST>         ::= IF <COND THEN> <STATEMENT*>
10   <K REG ASS>
11   <NULL ST>
12   <PROC DECL>          ::= <PROC HD6> <STATEMENT*>
13   <PROC ID>
14   <PROGRAM>
15   <SEG DECL>           ::= <SEG HEAD> BASE <K REG>
16   <SYN DC2>
17   <T CELL ASS>         ::= <T CELL> := <K REG>
18   <T DECL7>
19   <WHILE ST>           ::= <WHILE> <COND DO> <STATEMENT*>
```

Figure 1.—Documentation units.

and such that G^* contains syntactic entities we want to classify as documentation units and use to define lines.

The basis for the transformation of G into G^* is a finite number of elementary steps as follows. If X is any finite sequence of tokens and/or syntactic entities which occurs as part of the right side of a production rule in a grammar G^k, and $\langle A \rangle$ is not a syntactic entity in G^k, we can define a new production $\langle A \rangle ::= X$ and substitute $\langle A \rangle$ for X in the right side of any rule we please in G^k, to get a grammar G^{k+1}. It is clear that $L(G^{k+1}) = L(G^k)$ by this construction. Then, we consider a sequence

$$G = G^0, G^1, \ldots, G^n = G^*$$

where n is the (finite) number of additional syntactic entities we want to be defined in G^* which are not in G.

We note that even though additional syntactic entities can easily be introduced in a grammar while retaining the identical language, the question of keeping it a precedence grammar (ref. 2) is a delicate matter. This general point is not pursued here. However, we use only transformations which label the entire right side of a rule; in this case the grammar obviously retains its precedence properties.

In what follows, the grammar G is augmented to G^* just to provide a basis for invoking additional interpretation rules which define documentation files and generate questions. It will also be apparent that the same device can be useful in extending syntax processing beyond documentation to questions of execution control and dynamic storage allocation in multiprogramming operating systems. For example, better use of core may arise if core is allocated to the machine code responding to syntactic entities such as "for statements" and "while statements" rather than simply arbitrary "pages" of machine code which may break up such natural units of execution.

DOCUMENTATION UNITS

We classify as a documentation unit any right-hand side of a rule which reduces to one of the following syntactic entities in reference 1:

⟨SIMPLE STATEMENT⟩
⟨STATEMENT⟩
⟨DECL⟩
⟨PROGRAM⟩

There are 19 such documentation units given in figure 1. If the right-hand side is already defined in G, it is used directly. Otherwise, a new syntactic entity is defined, with the understanding that G is augmented by each such definition, as described above.

In effect, this classification of documentation rules is a convenience for identifying productions whose recognition in an analysis corresponds to having additional interpretation rules that deal with documentation processing.

Given a PL360 program, we consider every realization of such documentation units, which can be structured on the basis of syntactic membership, as follows. A documentation unit is a member of a second documentation unit if its program text is a subset of the program text of the second. It is an immediate member if it is not a member of any third documentation unit, itself a member of the second.

The relation of immediate membership defines a nested structure of documentation units in a program, beginning with the program itself as the highest level documentation unit and continuing through "blocks," "compound statements," etc., to "single declarations" and "single statements" at the lowest levels. This nested structure can also be described as a rooted tree, with the program as the root, and other documentation units as remaining intermediate and endpoint nodes in the tree.

Notice any given statement or declaration may be included in the program text of many documentation units. In fact, every documentation unit is a member of the program and of every other documentation unit whose text contains it.

SYNTAX-DEFINED PROGRAM LISTINGS

Next, we consider the question of listing programs written in PL360 in a standard way for readability and referencing during programmer interrogation and later examination. When programmers make an informal effort to arrange their programs for readability, they typically start each documentation unit, as defined above, on a new line and use indentation to correspond in a general way with syntactical nesting in the program. We recognize that the problem is a subjective one, but we give a syntax-defined listing algorithm which is believed to satisfy the intuitive intentions observed in informal programming efforts.

For the purpose of typographical listing, we partition a PL360 program or procedure into a string of substrings. Each substring is to be a printed line, and the string of lines constitutes a listing of the program. Associated with each line are two numbers: one which specifies its order in the program or procedure, and one which corresponds to the indentation (or starting column) of the line. If a line exceeds the width of paper available, its continuation is further indented a standard amount.

The partition of a PL360 program or procedure into lines is defined by marking the starting text for each documentation unit, and each label, BEGIN, END, ELSE, and . (dot) symbol. The lines are numbered consecutively. The indentation number is the level of nesting of the documentation unit it begins, if any, based on syntactic membership as described above. The only lines not beginning a new documentation unit are BEGIN (in CASE statements), END, ELSE, and . (dot). In each case they are indented according to the level of the documentation unit which they help define. Labels are given the indentation level of the program or procedure being listed.

To refer to a line from outside a procedure, we qualify the line numbers with the procedure name. While the concept of program is defined in PL360, no provision is made for naming a program in the syntax.

```
<BLOCK>
    Q1  PURPOSE OF BLOCK (COORDINATES)?
    S1  BLOCK (COORDINATES) IS TO (RESPONSE).
<CASE ST>
    Q1  PURPOSE OF CASE STATEMENT (COORDINATES)?
    S1  CASE STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  CASE SELECTED AT (COORDINATE)?
    S2  CASE SELECTED AT (COORDINATE) IS (RESPONSE).
<FOR ST>
    Q1  PURPOSE OF FOR STATEMENT (COORDINATES)?
    S1  FOR STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  FOR CONDITION AT (COORDINATE)?
    S2  FOR CONDITION AT (COORDINATE) IS TO (RESPONSE).
<FUNC DECL?>
    Q1  FUNCTION OPERATION AT (COORDINATE)?
    S1  FUNCTION OPERATION AT (COORDINATE) IS TO (RESPONSE).
<FUNC ID>
    Q1  PURPOSE OF FUNCTION STATEMENT AT (COORDINATE)?
    S1  FUNCTION STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<FUNC ST>
    Q1  PURPOSE OF FUNCTION STATEMENT AT (COORDINATE)?
    S1  FUNCTION STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<GOTO ST>
    Q1  GO TO WHERE AT (COORDINATE)?
    S1  AT (COORDINATE) CONTROL GOES TO (RESPONSE).
<IF THEN ELSE ST>
    Q1  PURPOSE OF IF THEN ELSE STATEMENT (COORDINATES)?
    S1  IF THEN ELSE STATEMENT AT (COORDINATES) IS TO (RESPONSE).
    Q2  IF CONDITION AT (COORDINATE)?
    S2  IF CONDITION AT (COORDINATE) TESTS (RESPONSE).
<IF THEN ST>
    Q1  PURPOSE OF IF THEN STATEMENT (COORDINATES)?
    S1  IF THEN STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  IF CONDITION AT (COORDINATE)?
    S2  IF CONDITION AT (COORDINATE) TESTS (RESPONSE).
<K REG ASS>
    Q1  VALUE OF (<ID>) AT (COORDINATE)?
    S1  VALUE OF (<ID>) AT (COORDINATE) IS (RESPONSE).
<NULL ST>
    Q1  PURPOSE OF NULL STATEMENT AT (COORDINATE)?
    S1  NULL STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<PROC DECL>
    Q1  AUTHOR OF PROCEDURE (<ID>)?
    S1  AUTHOR OF PROCEDURE (<ID>) IS (RESPONSE).
    Q2  PURPOSE OF PROCEDURE?
    S2  PROCEDURE (<ID>) IS TO (RESPONSE).
    Q3  INITIAL DATA?
    S3  INITIAL DATA OF PROCEDURE (<ID>) IS (RESPONSES).
    Q4  PROCESSING LOGIC?
    S4  PROCESSING LOGIC OF PROCEDURE (<ID>) IS TO (RESPONSE).
    Q5  FINAL DATA?
    S5  FINAL DATA OF PROCEDURE (<ID>) IS (RESPONSE).
    Q6  REFERENCES?
    S6  REFERENCES FOR PROCEDURE (<ID>) ARE (RESPONSE).
<PROC ID>
    Q1  PURPOSE OF PROCEDURE STATEMENT AT (COORDINATE)?
    S1  PROCEDURE (<PROC ID>) AT (COORDINATE) IS TO (RESPONSE).
<PROGRAM>
    Q1  AUTHOR OF PROGRAM (<ID>)?
    S1  AUTHOR OF PROGRAM (<ID>) IS (RESPONSE).
    Q2  PURPOSE OF PROGRAM ?
    S2  PROGRAM (<ID>) IS TO (RESPONSE).
    Q3  INITIAL DATA?
    S3  INITIAL DATA OF PROGRAM (<ID>) IS (RESPONSE).
    Q4  PROCESSING LOGIC?
    S4  PROCESSING LOGIC OF PROGRAM (<ID>) IS TO (RESPONSE).
    Q5  FINAL DATA?
    S5  FINAL DATA OF PROGRAM (<ID>) IS (RESPONSE).
    Q6  REFERENCES?
    S6  REFERENCES FOR PROGRAM (<ID>) ARE (RESPONSE).
<SEQ DECL>
    NO QUESTION
    NO STATEMENT
<SYN DC2> (FOR EACH IDENTIFIER DECLARED)
    Q1  SYNONYM (<ID>) TO (<ID>) AT (COORDINATE)?
    S1  SYNONYM (<ID>) TO (<ID>) AT (COORDINATE) IS (RESPONSE).
<T CELL ASS>
    Q1  VALUE OF (<ID>) AT (COORDINATE)?
    S1  VALUE OF (<ID>) AT (COORDINATE) IS (RESPONSE).
<T DECL?>
    Q1  (<ID>) AT (COORDINATE)?
    S1  (<ID>) AT (COORDINATE) IS (RESPONSE).
<WHILE ST>
    Q1  PURPOSE OF WHILE STATEMENT (COORDINATES)?
    S1  WHILE STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  WHILE CONDITION AT (COORDINATE) ?
    S2  WHILE CONDITION AT (COORDINATE) TESTS (RESPONSE).
```

Figure 2.–Skeleton question/statements for documentation units.

For convenience, we introduce a new basic symbol PROGRAM and the redefinition

⟨PROGRAM⟩ ::=
PROGRAM ⟨ID⟩ ⟨STATEMENT⟩

which permits the naming of programs and reference to documentation units by line numbers, qualified by program names.

CANONICAL DATA FILE

For convenience in documentation processing, we define a canonical data file as consisting of a record for each documentation unit of a program or procedure declaration. Its function is not only to store relationships between various syntactic entities but also to provide data for driving interrogation, report generation, and query processing concerning the program or procedure. Each record describes three properties of the documentation unit: its coordinates in the program text, its syntactic type, and an identifier list. The coordinates are the first and the last lines of the documentation unit (which may be the same when text is contained in a single line). The syntactic type is the entity identified as a documentation unit in figure 1. The identifier list depends on the syntactic type–denoting identifiers which are declared, assigned values, used in assigning values, used in control logic, etc.

It is clear that a deeper syntactical structure, described only informally here, is relevant below the generic level of documentation unit. For example, the identifier list itself is definable in terms of productions within a documentation unit, and such productions determine whether each identifier is being declared, assigned a value, used in a computation, used in control logic, etc. Thus the additional interpretation rules required for documentation processing are distributed throughout the syntax, all the way down to the identifier level, but are not discussed in detail now.

SYNTAX-DIRECTED INTERROGATION AND RESPONSE EDITING

We consider an automatic interrogation process, which uses the canonical data file to complete prestored skeleton questions with program text coordinates and/or identifiers. The interrogation process proceeds through the file, a record at a time, and generates a series of questions from each record, depending on the syntactic type and identifier list found therein. The responses to such questions, made by the programmer, are indexed to the records which generated them.

A set of skeleton questions associated with different documentation units in PL360 is displayed in figure 2. At the end of each interrogation, the programmer is given a final opportunity to volunteer any additional information.

Associated with each skeleton question in figure 2 is a skeleton statement which contains the programmer's response to that question as one of its parts. These statements, filled in with responses and other data from the canonical data file, as shown, represent basic unit messages which can be assembled into reports and query replies.

The construction of skeleton questions and skeleton statements to elicit and edit programmer responses is a substantial and still open problem. It is evident that careless questioning can bury programmers in questionnaires and alienate them to the whole idea. Limited experience (refs. 3 and 4) has indicated that skeleton questions should be terse and highly selective. An involved question, which seems reasonable to read once or twice, can have a very negative effect on a responder when repeated many times, even though this kind of question requires no more effort to answer than a terse one. Thus a first principle in question construction is that the burden of understanding what the question means must be put into a separate orientation course, outside the interrogation itself, and the questionnaires must be kept as short as practicable.

A second principle in question formation is that program text itself must be depended upon for later programmer reference. The questions and responses are intended to illuminate the program text, not to replace it. Otherwise, questions become too involved with points in plain sight in the program text.

Similarly, the order of questioning is also important. Some experience indicates that a "top-down" sequence is a better basis for questioning than "bottom-up." Fortunately, due to the structure of PL360, interrogating documentation units in the order in which their starting text appears gives a top-down approach, which seems easy to follow and reference from both syntactic and typographical viewpoints.

It has been suggested that the matter of question formation might be related to the problem of proving the correctness of programs. Naur (ref. 5) discusses an approach to proving the correctness of programs by "general snapshots," e.g., the state of all variables at various points in programs. These general snapshots could be defined at the entries to and exits from documentation units. This raises the possibility of forming such questions as: "What variables can be modified in this documentation unit?" and "What relationships between the variables must hold (a) on entry to or (b) on exit from this documentation unit?"

At the moment, no suitable way of forming such deeper questions for automatic interrogation is known. But this is an area where future progress may be possible.

DOCUMENTATION PRODUCTS

As already noted, two principal documentation products are—

Documentation reports: complete descriptions, in a prescribed format, of programs or procedures.

Query replies: partial reports in response to queries made by programmers familiar with programs or procedures to probe specific details.

It is to be noted that both interrogation and query reply processing lend themselves to conversational techniques (ref. 4). The canonical data file can be used to drive a conversational interrogation of a programmer quite directly. Similarly, the same file, with an associated file of indexed programmer responses, can be used to generate "computer-assisted instruction courses" automatically when the subjects are particular PL360 programs or procedures.

It should be emphasized that the documentation discussed is addressed to a programmer who understands PL360 and will be reading the PL360 text concurrently. The documentation products are not intended to replace this text as the ultimate authority of what the program does. Rather these products are intended to supplement the program text with perspective, motivation, identifier meanings, processing rationale, etc. In this way it is expected to increase the power and precision with which a programmer can deal with the program text, to modify it, to verify its functional logic, and to assure the integrity of a programming system containing it.

The documentation products will not themselves fill needs of higher level documentation related to user directions, instruction manuals, etc. However, technical writers concerned with such higher level documentation should find these products extremely useful as source material.

DOCUMENTATION REPORTS

We define a standard documentation report with three parts:

(1) Program text
(2) Edited responses
(3) Cross-references

The program text is the relisted, labeled text used in interrogation. The typographical arrangement of this relisting itself shows the overall syntactic structure of the program and/or procedures.

The edited responses, listed in the same order as the questions which generated them, proceed through the text in a systematic way so that one can refer back and forth between the relisted text and the responses efficiently in reading them together. It is expected that the program text and edited responses will be read together by programmers. It would be feasible to intersperse the responses, as comments, in the text, but it seems more desirable to treat them as separate documents with easy interference facilities.

In fact, as a programmer becomes more familiar with the details of a program, the presence of extensive comments tends to inhibit the visual perception of program structure

and logic: first, by simply taking up space and expanding the size of material to be looked at; and second, by interrupting and masking typographical features corresponding to the syntactical structure of the program.

The cross-references assemble identifer, function, and procedure usage into cross-reference tables. Identifier usage in the text is categorized into "declared," "assigned," "used in assignments," and "used in control." It is expected that these cross-references serve most of a programmer's needs for evaluating and/or modifying small programs or procedures; for example, to assure that all implications of a changed data declaration are accounted for.

Note that such cross-references can be assembled directly by interpretation rules during program analysis at the time various productions are recognized but then are referred to only informally here.

One particular use of cross-references in PL360 of some potential importance is the recognition of commonality of data references. In particular, the use of identifiers synonymous with hardware registers, which add considerably to the readability of PL360 text, can be found with the aid of such cross-references.

QUERY REPLIES

It is possible to generate a documentation report for any size system of programs or procedures, of course, as a sequence of documentation reports of all its component procedures and programs. However, where documentation reports for a small procedure can be examined rather easily for any information in it, the human eye and mind cannot take in the scope and details of a large system so readily. Thus simply listing a documentation report of a large system, while perhaps of value as a hard-copy reference, is still unsatisfactory for a programmer seeking to understand, modify, or augment a procedure interacting with many other parts of the system. This may be even more critical for a system manager, who is trying to verify the correctness of a new procedure and to assure that no ill effects occur in the system in accepting that new procedure.

This very problem has motivated the foregoing acquisition of documentation as responses to specific questions so that the documentation can be indexed down to the statement and identifer level. Thus the documentation in a large system can be enhanced by the capability for automatic selective retrieval and analysis of documentation. In this sense, the problem of a programmer is not so different from other information systems where data must be stored for retrieval from many points of interest.

A query language for accessing the type of data in these documentation files can be readily imagined and is not defined in detail here. Its output could simply be a selection of edited responses, as defined above. As already noted, such a query capability would lend itself well to conversational methods of programmer access to the documentation. Its capabilities should include, for any given documentation unit, finding identifier usages, extracting "purpose of" responses for all its members, identifying all branch points, and locating all references to keywords in responses.

PROGRAMMER ADAPTATION

In the final analysis, it is expected that the important issues in making such a syntax-directed documentation process effective will be the soundness of the structural approach,

```
PROCEDURE MAGICSQUARE (R6);
COMMENT  THIS PROCEDURE ESTABLISHES A MAGIC SQUARE OF ORDER N, IF N IS
   ODD AND 1 < N < 16. X IS THE MATRIX IN LINEARIZED FORM. REGISTERS
   R0...R6 ARE USED, AND REGISTER R0 INITIALLY CONTAINS THE PARAMETER
   N. ALGORITHM 118 COMM. ACM 5 (AUG. 1962) ;
BEGIN SHORT INTEGER NSQR;
   INTEGER REGISTER N SYN R0, I SYN R1, J SYN R2, XX SYN R3,
   IJ SYN R4, K SYN R5;
   NSQR := N; R1 := N * NSQR; NSQR := R1;
   I := N + 1 SHRL 1; J := N;
   FOR K := 1 STEP 1 UNTIL NSQR DO
   BEGIN XX := I SHLL 6; IJ := J SHLL 2 + XX; XX := X(IJ);
      IF XX ¬= 0 THEN
      BEGIN I := I - 1; J := J - 2;
         IF I < 1 THEN I := I + N;
         IF J < 1 THEN J := J + N;
         XX := I SHLL 6; IJ := J SHLL 2 + XX;
      END;
      X(IJ) := K;
      I := I + 1; IF I > N THEN I := I - N;
      J := J + 1; IF J > N THEN J := J - N;
   END;
END
```

Figure 3.—Procedure Magicsquare (ref. 1, p. 53).

```
 1  PROCEDURE MAGICSQUARE (R6);
 2    BEGIN
 3      SHORT INTEGER NSQR;
 4      INTEGER REGISTER N SYN R0, I SYN R1, J SYN R2, XX SYN R3,
        IJ SYN R4, K SYN R5;
 5      NSQR := N;
 6      R1 := N * NSQR;
 7      NSQR := R1;
 8      I := N + 1 SHRL 1;
 9      J := N;
10      FOR K := 1 STEP 1 UNTIL NSQR DO
11        BEGIN
12          XX := I SHLL 6;
13          IJ := J SHLL 2 + XX;
14          XX := X(IJ);
15          IF XX ¬= 0 THEN
16            BEGIN
17              I := I - 1;
18              J := J - 2;
19              IF I < 1 THEN
20                I := I + N;
21              IF J < 1 THEN
22                J := J + N;
23              XX := I SHLL 6;
24              IJ := J SHLL 2 + XX;
25            END;
26          X(IJ) := K;
27          I := I + 1;
28          IF I > N THEN
29            I := I - N;
30          J := J + 1;
31          IF J > N THEN
32            J := J - N;
33        END;
34    END
```

Figure 4.—Syntax-defined listing of Magicsquare.

rather than niceties of question phrasing or report formation. This is because programmers, as human beings, have a large capacity to adapt to matters of English usage but a small capacity to deal with extended program syntax structures in detail.

In the interrogation process, programmers will soon learn how to phrase their responses gracefully in matters of English usage such as parts of speech and tense simply by examining the edited responses which their answers generate. Also, they will learn how the details of their rationale should be allocated among responses by experience in interrogation and by examining the resulting documentation reports. It will still take ability to document programs, but an ability which is adapted to the automatic process being used to acquire and dispense the documentation.

For example, a programmer new to the process may respond to a question about a block by going into the details of statements inside the block. After going through several interrogations and realizing he will be questioned about the included statements later anyway, he will learn to confine his response about the block to the block as a unit. Similarly, by learning that conditions for branching IF statement will be taken up separately, a programmer, following the treatment of the IF statement as a unit, will address his response to the IF statement itself.

In using the documentation of others, a programmer, from his own experience as an originating programmer, will be aware of the questions which generated the responses. He will know, simply by examining program text himself, what questions were asked about any documentation unit or identifier he may be interested in and where they were asked. Thus he can exert considerable intelligence in selective queries of documentation files.

AN EXAMPLE

Figures 3 to 9 simulate the foregoing methods on a sample PL360 procedure, found in reference 1, showing the relisting and interrogation, the canonical data file, a set of responses, a documentation report, and, finally, a set of query replies.

```
COOR-    DUC.    IDENTIFIERS
DINATES  UNIT
1,34      12     MAGICSQUARE, R6
2,34       1
3,3       18     NSQR
4,4       16     N, I, J, XX, IJ, K
5,5       17     NSQR, N
6,6       10     R1, N, NSQR
7,7       17     NSQR, R1
8,8       10     I, N
9,9       10     J, N
10,33      5     K, NSQR
11,33      1
12,12     10     XX, I
13,13     10     IJ, J, XX
14,14     10     XX, X(IJ)
15,25      9     XX
16,25      1
17,17     10     I, I
18,18     10     J, J
19,20      9     I
20,20     10     I, I, N
21,22      9     J
22,22     10     J, J, N
23,23     10     XX, I
24,24     10     IJ, J, XX
26,26     17     X, IJ, K
27,27     10     I, I
28,29      9     I, N
29,29     10     I, I, N
30,30     10     J, J
31,32      9     J, N
32,32     10     J, J, N
```

Figure 5.–Canonical data of Magicsquare.

```
FILE KEY  QUESTION

1,34,1    AUTHOR OF PROCEDURE MAGICSQUARE?
1,34,2    PURPOSE OF PROCEDURE MAGICSQUARE?
1,34,3    INITIAL DATA?
1,34,4    PROCESSING LOGIC?
1,34,5    FINAL DATA?
1,34,6    REFERENCES?
2,34,1    PURPOSE OF BLOCK 2,34 ?
3,3,1     NSQR AT 3 ?
4,4,1     N AT 4 ?
4,4,2     I AT 4 ?
4,4,3     J AT 4 ?
4,4,4     XX AT 4 ?
4,4,5     IJ AT 4 ?
4,4,6     K AT 4 ?
5,5,1     VALUE OF NSQR AT 5 ?
6,6,1     VALUE OF R1 AT 6 ?
7,7,1     VALUE OF NSQR AT 7 ?
8,8,1     VALUE OF I AT 8 ?
9,9,1     VALUE OF J AT 9 ?
10,33,1   PURPOSE OF FOR STATEMENT 10,33 ?
10,33,2   FOR CONDITION AT 10 ?
11,33,1   PURPOSE OF BLOCK 11,33 ?
12,12,1   VALUE OF XX AT 12 ?
13,13,1   VALUE OF IJ AT 13 ?
14,14,1   VALUE OF XX AT 14 ?
15,25,1   PURPOSE OF IF THEN STATEMENT 15,25 ?
15,25,2   IF CONDITION AT 15 ?
16,25,1   PURPOSE OF BLOCK 16,25 ?
17,17,1   VALUE OF I AT 17 ?
18,18,1   VALUE OF J AT 18 ?
19,20,1   PURPOSE OF IF THEN STATEMENT 19,20 ?
19,20,2   IF CONDITION AT 19 ?
20,20,1   VALUE OF I AT 20 ?
21,22,1   PURPOSE OF IF THEN STATEMENT 21,22 ?
21,22,2   IF CONDITION AT 21 ?
22,22,1   VALUE OF J AT 22 ?
23,23,1   VALUE OF XX AT 23 ?
24,24,1   VALUE OF IJ AT 24 ?
26,26,1   VALUE OF X(IJ) AT 26 ?
27,27,1   VALUE OF I AT 27 ?
28,29,1   PURPOSE OF IF THEN STATEMENT 28,29 ?
28,29,2   IF CONDITION AT 28 ?
29,29,1   VALUE OF I AT 29 ?
30,30,1   VALUE OF J AT 30 ?
31,32,1   PURPOSE OF IF THEN STATEMENT 31,32 ?
31,32,2   IF CONDITION AT 31 ?
32,32,1   VALUE OF J AT 32 ?
1,34,7    ANY FURTHER COMMENTS ?
```

Figure 6.–Syntax-defined interrogation for Magicsquare.

Figure 3 is a PL360 procedure named by Magicsquare, just as formulated by Wirth (ref. 1), including the typography. This procedure, adapted from an ALGOL procedure published in the Algorithm department of *Communications* of the ACM (ref. 6), builds magic squares of odd order n when $1 < n < 16$.

Figure 4 is a syntax-defined and labeled relisting of the same PL360 procedure Magicsquare, less comments, with its typography determined by the rules already given for recognizing lines and their indentation. This relisting is independent of the typography of the program text in figure 3. It is expected that such a standard yet flexible form of program text will, in itself, help programmers read each other's programs.

Figure 5 shows the contents of the canonical data file generated by procedure Magicsquare. All further interrogation, response editing, and other documentation processing will use this canonical data file and not the program text. This particular file contains 31 records with some 157 separate items of data in them: two coordinates, a syntactic type, and an average of about two identifiers per record.

Figure 6 gives the syntax-directed interrogation of Magicsquare, using the canonical data file and the skeleton questions of figure 2. There are 48 questions in all, which refer to the coordinates of the relisted program text and represent a systematic coverage of the text. A final question gives a programmer an opportunity to volunteer additional information not already solicited by the previous questions.

Figure 7 contains a set of responses to the interrogation of figure 6. There is a file key associated with each question, which is used to label responses so that they may be indexed to the proper questions. The author has presumed to speak for "programmer Wirth" in constructing these responses.

Figure 8 provides a resulting documentation report in the three sections described already: source code, edited responses, and cross-references. For a short procedure or program such as this one, it is expected that a documentation report itself will be sufficient to allow a programmer to find out anything he wants to know about the procedure or program.

```
FILE KEY  RESPONSE

1,34,1    NIKLAUS WIRTH, STANFORD UNIVERSITY, DECEMBER 20, 1966.
1,34,2    ESTABLISH A MAGIC SQUARE OF ORDER N, IF N IS ODD AND 1 < N < 16.
1,34,3    THE ORDER, N, OF THE MAGIC SQUARE DESIRED.
1,34,4    FILL SQUARE MATRIX WITH SUCCESSIVE INTEGERS ALONG CERTAIN DIAGONALS
          AND THEIR EXTENSIONS TO ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO
          BE FILLED IS ASSUMED TO CONTAIN ALL ZEROES INITIALLY.
1,34,5    THE MAGIC SQUARE X AS A MATRIX IN LINEARIZED FORM.
1,34,6    ALGORITHM 116, COMM ACM, AUGUST 1962, P 436; M. KRAITCHIK,
          MATHEMATICAL RECREATIONS, P 149.
2,34,1    CARRY OUT THE PROCEDURE MAGICSQUARE.
3,3,1     THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
4,4,1     THE ORDER (NUMBER OF ROWS AND COLUMNS) OF THE MAGIC SQUARE.
4,4,2     THE ROW INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
4,4,3     THE COLUMN INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE MAGIC
          SQUARE.
4,4,4     INTERMEDIATE VALUE IN X OFFSET CALCULATION AND TO TEST X VALUE FOR
          ZERO.
4,4,5     THE X OFFSET FOR ROW I, COLUMN J OF MAGIC SQUARE.
4,4,6     THE NEXT INTEGER VALUE GOING INTO MAGIC SQUARE.
5,5,1     INTERMEDIATE VALUE N FOR NSQR.
6,6,1     TEMPORARY STORAGE OF NSQR
7,7,1     FINAL VALUE OF NSQR, THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
8,8,1     INITIAL VALUE FOR I.
9,9,1     INITIAL VALUE FOR J.
10,33,1   FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   STEP K THROUGH INTEGERS FROM 1 TO NSQR, WHICH WILL APPEAR IN THE
          MAGIC SQUARE.
11,33,1   FIND CORRECT LOCATION IN MAGIC SQUARE FOR INTEGER K.
12,12,1   X OFFSET FOR ROW I OF MAGIC SQUARE.
13,13,1   X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
14,14,1   CURRENT VALUE OF POINT I, J IN MAGIC SQUARE.
15,25,1   BEGIN NEW DIAGONAL IF CURRENT DIAGONAL IS ALREADY FILLED.
15,25,2   IS DIAGONAL FILLED (AN INTEGER ALREADY STORED AT POINT I,J)?
16,25,1   FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE FILLED.
17,17,1   NEW ROW INDEX OF STARTING LOCATION.
18,18,1   NEW COLUMN INDEX OF STARTING LOCATION.
19,20,1   RESTORE ROW INDEX TO CORRECT RANGE, IF NECESSARY.
19,20,2   IS ROW INDEX OUT OF RANGE?
20,20,1   ROW INDEX IN CORRECT RANGE.
21,22,1   RESTORE COLUMN INDEX TO CORRECT RANGE, IF NECESSARY.
21,22,2   IS COLUMN INDEX IN CORRECT RANGE ?
22,22,1   COLUMN INDEX IN CORRECT RANGE.
23,23,1   X OFFSET FOR ROW I OF MAGIC SQUARE.
24,24,1   X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
26,26,1   FINAL INTEGER VALUE AT POINT I, J IN MAGIC SQUARE.
27,27,1   ROW INDEX STEPPED ALONG DIAGONAL.
28,29,1   RESTORE ROW INDEX TO CORRECT RANGE, IF NECESSARY.
28,29,2   IS ROW INDEX IN CORRECT RANGE?
29,29,1   ROW INDEX IN CORRECT RANGE.
30,30,1   COLUMN INDEX STEPPED ALONG DIAGONAL.
31,32,1   RESTORE COLUMN INDEX TO CORRECT RANGE, IF NECESSARY.
31,32,2   IS COLUMN INDEX IN CORRECT RANGE?
32,32,1   COLUMN INDEX IN CORRECT RANGE.
1,34,7    NO.
```

Figure 7.—Interrogation responses for Magicsquare.

MAGICSQUARE PROGRAM TEXT

```
1  PROCEDURE MAGICSQUARE (R6);
2    BEGIN
3      SHORT INTEGER NSQR;
4      INTEGER REGISTER N SYN R0, I SYN R1, J SYN R2, XX SYN R3,
         IJ SYN R4, K SYN R5;
5      NSQR := N;
6      R1 := N * NSQR;
7      NSQR := R1;
8      I := N + 1 SHRL 1;
9      J := N;
10     FOR K := 1 STEP 1 UNTIL NSQR DO
11       BEGIN
12         XX := I SHLL 6;
13         IJ := J SHLL 2 + XX;
14         XX := X(IJ);
15         IF XX ¬= 0 THEN
16           BEGIN
17             I := I - 1;
18             J := J - 2;
19             IF I < 1 THEN
20               I := I + N;
21             IF J < 1 THEN
22               J := J + N;
23             XX := I SHLL 6;
24             IJ := J SHLL 2 + XX;
25           END;
26         X(IJ) := K;
27         I := I + 1;
28         IF I > N THEN
29           I := I - N;
30         J := J + 1;
31         IF J > N THEN
32           J := J - N;
33     END;
34   END
```

MAGICSQUARE EDITED RESPONSES

```
FILE KEY  EDITED RESPONSE

1,34,1    AUTHOR OF PROCEDURE MAGICSQUARE IS NIKLAUS WIRTH, STANFORD UNIVERSITY.
          DECEMBER 20, 1966.
1,34,2    PROCEDURE MAGICSQUARE IS TO ESTABLISH A MAGIC SQUARE OF ORDER N, IF N
          IS ODD AND 1 < N < 26.
1,34,3    INITIAL DATA OF PROCEDURE MAGICSQUARE IS THE ORDER, N, OF THE MAGIC
          SQUARE DESIRED.
1,34,4    PROCESSING LOGIC OF PROCEDURE MAGICSQUARE IS TO FILL SQUARE MATRIX
          WITH SUCESSIVE INTEGERS ALONG CERTAIN DIAGONALS AND THEIR EXTENSIONS
          TO ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO BE FILLED IS ASSUMED
          TO CONTAIN ALL ZEROES INITIALLY.
1,34,5    FINAL DATA OF PROCEDURE MAGICSQUARE IS THE MAGIC SQUARE X AS A MATRIX
          IN LINEARIZED FROM.
1,34,6    REFERENCES FOR MAGICSQUARE ARE ALGORITHM 116, COMM ACM, AUGUST 1962,
          P 436; M. KRAITCHIK, MATHEMATICAL RECREATIONS, P 149.
2,34,1    BLOCK 2,34 IS TO CARRY OUT THE PROCEDURE MAGICSQUARE.
3,3,1     NSQR AT 3 IS THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
4,4,1     N AT 4 IS THE ORDER (NUMBER OF ROWS OR COLUMNS) OF THE MAGIC SQUARE.
4,4,2     I AT 4 IS THE ROW INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE
          MAGIC SQUARE.
4,4,3     J AT 4 IS THE COLUMN INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE
          MAGIC SQUARE.
4,4,4     XX AT 4 IS INTERMEDIATE VALUE IN X OFFSET CALCULATION AND TO TEST X
          VALUE FOR ZERO.
4,4,5     IJ AT 4 IS THE X OFFSET FOR ROW I, COLUMN J OF MAGIC SQUARE.
4,4,6     K AT 4 IS THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
5,5,1     VALUE OF NSQR AT 5 IS INTERMEDIATE VALUE FOR NSQR.
6,6,1     VALUE OF R1 AT 6 IS TEMPORARY STORAGE OF NSQR.
7,7,1     VALUE OF NSQR AT 7 IS FINAL VALUE OF NSQR, THE NUMBER OF ENTRIES IN
          THE MAGIC SQUARE.
8,8,1     VALUE OF I AT 8 IS INITIAL VALUE FOR I.
9,9,1     VALUE OF J AT 9 IS INITIAL VALUE FOR J.
10,33,1   FOR STATEMENT 10,33 IS TO FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR,
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
11,33,1   BLOCK 11,33 IS TO FIND CORRECT LOCATION IN MAGIC SQUARE FOR INTEGER K.
12,12,1   VALUE OF XX AT 12 IS X OFFSET FOR ROW I OF MAGIC SQUARE.
13,13,1   VALUE OF IJ AT 13 IS X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
14,14,1   VALUE OF XX AT 14 IS CURRENT VALUE OF POINT I, J IN MAGIC SQUARE.
15,25,1   IF THEN STATEMENT 15,25 IS TO BEGIN NEW DIAGONAL IF CURRENT DIAGONAL
          IS ALREADY FILLED.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J)?
16,25,1   BLOCK 16,25 IS TO FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE
          FILLED.
17,17,1   VALUE OF I AT 17 IS NEW ROW INDEX OF STARTING LOCATION.
18,18,1   VALUE OF J AT 18 IS NEW COLUMN INDEX OF STARTING LOCATION.
19,20,1   IF THEN STATEMENT 19,20 IS TO RESTORE ROW INDEX TO CORRECT RANGE, IF
          NECESSARY.
19,20,2   IF CONDITION AT 19 TESTS IS ROW INDEX OUT OF RANGE?
20,20,1   VALUE OF I AT 20 IS ROW INDEX IN CORRECT RANGE.
21,22,1   IF THEN STATEMENT 21,22 IS TO RESTORE COLUMN INDEX TO CORRECT RANGE,
          IF NECESSARY.
21,22,2   IF CONDITION AT 21 TESTS IS COLUMN INDEX IN CORRECT RANGE?
22,22,1   VALUE OF J AT 22 IS COLUMN INDEX IN CORRECT RANGE.
23,23,1   VALUE OF XX AT 23 IS X OFFSET FOR ROW I OF MAGIC SQUARE.
24,24,1   VALUE OF IJ AT 24 IS X OFFSET FOR ROW I OF COLUMN J OF MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I, J IN MAGIC
          SQUARE.
27,27,1   VALUE OF I AT 27 IS ROW INDEX STEPPED ALONG DIAGONAL.
28,29,1   IF THEN STATEMENT 28,29 IS TO RESTORE ROW INDEX TO CORRECT RANGE, IF
          NECESSARY.
28,29,2   IF CONDITION AT 28 TESTS IS ROW INDEX IN CORRECT RANGE?
29,29,1   VALUE OF I AT 29 IS ROW INDEX IN CORRECT RANGE.
30,30,1   VALUE OF J AT 30 IS COLUMN INDEX STEPPED ALONG DIAGONAL.
31,32,1   IF THEN STATEMENT 31,32 IS TO RESTORE COLUMN INDEX TO CORRECT RANGE,
          IF NECESSARY.
31,32,2   IF CONDITION AT 31 TESTS IS COLUMN INDEX IN CORRECT RANGE?
32,32,1   VALUE OF J AT 32 IS COLUMN INDEX IN CORRECT RANGE.
1,34,7    NO FURTHER COMMENTS.

MAGICSQUARE CROSS REFERENCES

     DATA CROSS REFERENCES

          I: DC 4; AS 6,8,17,20,27,29; UA 7,12,17,20,23,27,29; UC 19,28;
          IJ: DC 4; AS 13,24; UA 14,26;
          J: DC 4; AS 9,18,22,30,31; UA 13,19,22,24,30,31; UC 21,31;
          K: DC 4; AS 10; UA 10,26; UC 10;
          MAGICSQUARE: DC 1;
          N: DC 4; UA 5,6,8,9,20,22,29,32; UC 28,31;
          NSQR: DC 3; AS 5,7; UA 6; UC 10;
          R0: UA 5,6,8,9,20,22,29,32; UC 28,31;
          R1: AS 6,8,17,20,27,29; UA 7,12,17,20,23,27,29; UC 19,28;
          R2: AS 9,18,22,30,31; UA 13,19,22,24,30,31; UC 21,31;
          R3: AS 12,14,23; UA 13,24; UC 15;
          R4: AS 13,24; UA 14,26;
          R5: AS 10; UA 10,26; UC 10;
          R6: UC 1;
          X(IJ): AS 26; UA 14;
          XX: DC 4; AS 12,14,23; UA 13,24,; UC 15;

     FUNCTION CROSS REFERENCES

          NO FUNCTION CROSS REFERENCES.

     PROCEDURE CROSS REFERENCES

          NO PROCEDURE CROSS REFERENCES.
```

Figure 8.—Documentation report for Magicsquare.

```
   QUERY: ALL REFERENCES TO K

     QUERY REPLY:

4,4,6     K AT 4 IS THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
10,33,1   FOR STATEMENT 10,33 IS TO FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR.
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I,J IN MAGIC
          SQUARE.

   QUERY: ALL BRANCHES

     QUERY REPLY:
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR.
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J) ?
19,20,2   IF CONDITION AT 19 TESTS IS ROW INDEX OUT OF RANGE ?
21,22,2   IF CONDITION AT 21 TESTS IS COLUMN INDEX IN CORRECT RANGE ?
28,29,2   IF CONDITION AT 28 TEST IS ROW INDEX IN CORRECT RANGE ?
31,32,2   IF CONDITION AT 31 TESTS IS COLUMN INDEX IN CORRECT RANGE ?

   QUERY: ALL REFERENCES TO KEYWORD 'DIAGONAL' IN RESPONSES

     QUERY REPLY:
1,34,4    PROCESSING LOGIC OF PROCEDURE MAGICSQUARE IS TO FILL SQUARE MATRIX WITH
          SUCESSIVE INTEGERS ALONG CERTAIN DIAGONALS AND THEIR EXTENSIONS TO
          ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO BE FILLED IS ASSUMED TO
          CONTAIN ALL ZEROES INITIALLY.
15,25,1   IF THEN STATEMENT 15,25 IS TO BEGIN NEW DIAGONAL IF CURRENT DIAGONAL
          IS ALREADY FILLED.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J)?
16,25,2   BLOCK 16,25 IS TO FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE
          FILLED.
27,27,1   VALUE OF I AT 27 IS ROW INDEX STEPPED ALONG DIAGONAL.
30,30,1   VALUE OF J AT 30 IS COLUMN INDEX STEPPED ALONG DIAGONAL.

   QUERY: ALL USES IN ASSIGNMENTS OF IJ

     QUERY REPLY:

14,14,1   VALUE OF XX AT 14 IS CURRENT VALUE OF POINT I,J IN MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I,J IN MAGIC
          SQUARE.
```

Figure 9.—Some query replies for Magicsquare.

Figure 9 indicates how certain queries might be used to probe more specifically into the procedure via syntactic, identifier, or response keyword criteria. Note in each case a subset of the edited responses of a full documentation report is simply compiled according to a query condition.

In all these listings, the file keys have been listed to make the storage/retrieval process transparent. In practice, they could be suppressed in documentation reports and query replies.

ACKNOWLEDGMENTS

The author acknowledges useful suggestions from referees, particularly on some specifics of PL360 and on the automatic formation of questions. The relationship between proving the correctness of programs and the interrogation process was suggested by a referee.

REFERENCES

1. Wirth, N.: PL360, A Programming Language for the 360 Computers. *J. Ass. Computing Machinery* **15**: 37-74, Jan. 1968.
2. Wirth, N.; and Weber, H. Euler: A Generalization of ALGOL, and Its Formal Definition: Pt. I. *Commun. Ass. Computing Machinery* **9**(1): 13-23, Jan. 1966.
3. Mills, H. D.; and Dyer, M.: Evolutionary Systems for Data Processing. *IBM Real-Time Systems Seminar,* Nov. 1966, pp. 1-9.
4. Meadow, C. T.; and Waugh, D. V.: Computer Assisted Interrogation. *Proc. AFIPS 1966 Fall Joint Comput. Conf.* Vol. 29, Spartan Books, Inc., pp. 381-394.

5. Naur, P.: Proof of Algorithms by General Snapshots. *BIT* **6**: 310-316, 1966.
6. Collison, D. M.: Algorithm 118, Magic Square (Odd Order). *Commun. Ass. Computing Machinery* **5**(8): 456, Aug. 1962.

DISCUSSION

MEMBER OF THE AUDIENCE: Could you in the running program have asked some questions beforehand, such as what are the ranges of your variables, so that this could be incorporated into the program for error analysis? Could you also use this question-and-answer sort of thing for compiling optimization, so that you actually had a sort of interactive compiler? Do you think these kinds of things might be feasible?

DR. MILLS: Well, I think they probably can. I have not thought about them, but I think that what you say sounds reasonable. I really laid out a very austere kind of thing. It is easy for the mind to boggle at the idea of trying to do computer-assisted interrogation of almost any subject. The computer programs are particularly well structured. I mean we can actually define the syntax. But doing this in other areas may be far-fetched.

MEMBER OF THE AUDIENCE: How long would it take to develop this system?

DR. MILLS: Well, what I described here to you is a paper system because we do not have PL360. But I hope I can get a couple of graduate students to do this quickly.