INTRODUCTION

The XPOP programming system is a straightforward and practical means of implementing on a computer a great variety of languages—in other words, of writing a variety of compilers. The class of languages it can handle is not easy to characterize by syntactic form, since the system permits syntax specification to be varied freely from statement to statement in a program being scanned; the permitted class includes the best-known programming languages, as well as something closely approaching natural language. We believe that this distinguishes the XPOP processor from the syntax-directed compilers, although it shares with them the fundamental idea that the process of programming-language translation can be usefully generalized by a compiler to which source-language syntax is specified as a parameter.

This paper describes only the more novel features of XPOP; a fuller treatment is available elsewhere.

DISCUSSION

XPOP consists of two major parts: (1) a generalized skeleton-compiler that performs those functions common to all compilers, and (2) a battery of pseudo-operations for specifying the notation, operation repertoire, and compiling peculiarities of a desired programming language. The programmer creates the compiler for such a language not by programming it from scratch but by using the XPOP pseudo-operations to modify and extend XPOP itself, which then becomes the desired compiler.

The use of these facilities involves the creation by the programmer of functional units that superficially resemble the programmer-defined macro-instructions of, for example, IBMAP (and in fact include such macros as a subset), but whose effects may be radically different from those obtained by use of conventional macros. An XPOP macro does not necessarily generate coding; its possible effects are so varied that it can best be defined simply as an element of the source program that, when identified, causes the processor to take some specified action. That action may be any of the following:

1. The parameterization of XPOP's scanning routine to make it recognize, either for the remainder of the source program or within some more limited domain, a new notation
2. The compilation of coding for immediate or remote insertion into the object program
3. The immediate assembly and execution of any of the instructions compiled from a source-language statement.

* By "conventional macros" we mean the user-defined operators that some programming systems allow. The definition of a macro consists essentially of the assignment of a name to a block of coding, after which every appearance of that name as an operator causes the system to insert a copy of that coding into the object program.
(4) The preservation on cards and/or tape of the language description currently in use, in a condensed format that can be redigested by XPOP at tape speed when read back in; also the reading-in of such a language from a tape file created earlier in the same machine run or during an earlier run.

(5) The production by XPOP of a bug-finding tool called an XRAY—a highly specialized core-and-tape dump giving the programmer the tables and strings produced by the system in structured, interpreted, and captioned form.

In the illustrations of these features, some conventions that require explanation will be used. All programming examples offered are exact transcripts of the symbolic parts of actual XPOP listings. Lines prefixed by a dollar sign are records output by the processor as comments; these originate either as source-program statements printed out as comments for documentary purposes or as processor-generated messages notifying the programmer of errors or other conditions he should be aware of. No attempt is made to illustrate XPOP facilities by coding examples of any intrinsic value. The examples used are merely vehicles for the exhibition of those facilities and are therefore generally trivial in size and effect. The discussion that follows takes up the chief features of the system in the order of the five-point outline given earlier.

**Notation—Defining Pseudo-Operations**

Consider a macro, LOGSUM, created to store the logical sum of two boolean variables, A and B, in location C.

```plaintext
$LOGSUM MACRO A, B, C
$ CAL A
$ ORA B
$ SLW C
$ END
```

Having been defined, this macro may at once be called upon in XPOP’s standard form, which requires that the macro's name be immediately followed by the required parameters with commas separating these elements and the first blank terminating the statement. A standard-form call on LOGSUM would have this appearance and effect:

```plaintext
$ LOGSUM,ALPHA,BETA,GAMMA
    CAL ALPHA
    ORA BETA
    SLW GAMMA
```

Suppose we find standard-form notation unsatisfactory and want to call upon the function LOGSUM in the following form:

```plaintext
STORE INTO CELL 'C' THE LOGICAL SUM FORMED BY 'OR'ING THE BOOLEAN VARIABLES 'A' AND 'B'.
```

There are, from the XPOP programmer's viewpoint, four differences between the standard and the desired form:

1. The name of the function is no longer LOGSUM, but STORE.
2. The order in which parameters are expected by STORE differs from that of LOGSUM.
3. The punctuation required by the two forms differs; in standard form, the comma is the sole separator, blank the sole terminator. In the desired form, three kinds of separator are used:
   a. The one-character string 'blank'
   b. The two-character string 'blank-apostrophe'
   c. The two-character string 'apostrophe-blank'
   and one terminator
   a. The two-character string 'apostrophe-period'
4. The desired form contains several "noise words"—that is, character strings present for human convenience but which XPOP is to ignore.

In the following illustration, we use its pseudo-ops to teach XPOP the new statement form, then demonstrate that the lesson has been learned by offering it the new form as input and verifying that it produces the correct coding. An explanation of each pseudo-op used follows the illustration.
The definition of STORE with which the above illustration begins deals with the first two of the four differences noted between the desired and the standard statements. It causes XPOP to recognize STORE as an operator identical in effect to LOGSUM, and specifies that the parameter expected as the third by LOGSUM will be expected as the first by STORE. The pseudo-op CHPUNC (CHange PUNCTuation) deals with the third difference. Its first use, with blank variable field, erases all punctuation conventions from the system; the comma is no longer a separator nor is the blank a terminator. Having thus wiped the slate clean, CHPUNC is used again to specify the required punctuation. The variable field that follows this second CHPUNC may be read: "Three separators—the one-character string blank, the two-character string blank-apostrophe, and the two-character string apostrophe-blank; also one terminator—the two-character string apostrophe-period." (The additional punctuation specified by the third CHPUNC was introduced because the signal to XPOP that a statement is continued on the next card is the occurrence, at the end of each card's worth, of a separator immediately followed by a terminator; here the programmer wanted to use the string '...' for this purpose. A separate CHPUNC was necessary simply because the additional punctuation came as an afterthought.) The fourth and last difference is dealt with by means of the pseudo-op NOISE, which permits the programmer to specify character strings to be ignored by the processor. Since strings longer than six characters are taken as noise words if their first six characters are identical to any noise word, such strings as VARIABLE, VARIABLES, and VARIABILITY are effectively made noise words by the definition of 6VARIAB as an explicit noise word.
With these pseudo-ops given, XPOP has been taught the desired statement form, as proof of which it generates correctly parameterized coding when used as input. That statement was created, of course, only for illustrative purposes; few programmers would care to use so many words to generate three lines of machine-language coding. For an application in which documentation was an unusually important requirement, however, so elaborate a statement might serve a useful purpose—and real macros would average closer to 100 instructions than to 3.

The most important property of this technique for describing a notation to a processor, though, is the flexibility with which a notation so specified may be used. All that the XPOP programmer has explicitly defined is a number of individual words and punctuation marks, with no constraints on their combination; they may be used to form any statement that makes sense and conveys the necessary information to the processor. The programmer will often have a particular model statement in mind when specifying the vocabulary he wishes to use in calling for some function, but he will find that in implementing the model he has incidentally implemented an enormous number and variety of alternative forms.

If we add to our list of noise words the two strings OF and AS, we can use any of the following to generate the required coding:

(a) STORE INTO GAMMA THE SUM OF ALPHA AND BETA.
(b) STORE AS GAMMA THE LOGICAL SUM OF ALPHA AND BETA.
(c) STORE AS LOGICAL GAMMA THE SUM OF THE VARIABLES ALPHA AND BETA.
(d) STORE LOGICALLY INTO GAMMA 'ALPHA' AND 'BETA.'
(e) STORE GAMMA ALPHA BETA.
(f) LOGICALLY STORE INTO 'GAMMA' THE VARIABLES 'ALPHA' AND 'BETA.'
(g) INTO GAMMA STORE THE SUM OF ALPHA AND BETA.

As (f) and (g) indicate, both noise words and operands may precede the operator, provided only that they are not themselves mistakable for operators. If, for example, INTO were an operator as well as a noise word (such multiple roles are possible and sometimes useful), statement (g) would be misunderstood as a call on INTO. Excepting such uncommon cases, the operator and operands in a statement may float freely with respect to noise words, and the operator may float freely with respect to its operands; the sole constraint is that the operands must be given in the order specified when the operator was defined. Even this last constraint will be relaxed when the QWORD feature is fully implemented. A QWORD is a noise word that, like an English preposition, identifies the syntactic role of the word it precedes; its use enables the programmer to offer operands in an order independent of that specified when the operator is defined. Applied to the statement type dealt with so far, the QWORD feature might be used thus:

STORE MACRO $INTO$C,A,B
    CAL A
    ORA B
    SLW C
END

The string $INTO$C informs the system that if the QWORD "INTO" appears in a call on STORE, the first operand following it is to be taken as corresponding to the dummy variable C. The use of the QWORD would override the normal C,A,B order and enable the user of STORE to write, as another alternative:

(h) LOGICALLY STORE THE SUM OF ALPHA AND BETA INTO GAMMA.

Practically all notation-defining pseudo-ops may be used within macros as well as outside them, and the difference in location determines whether the conventions thereby established are 'local' or 'global.' If such pseudo-ops are given at the beginning of a macro definition that includes some non-pseudo-op lines as well, they are taken as local in effect. They will temporarily augment or supersede any notational conventions already established, and be
nullified when the macro within which they were found has been fully expanded. 'Local' notation-defining pseudo-ops will be put into effect in time to govern the scan of the very statement that calls on their containing macro. Such internally defined statements need respect the earlier conventions only to the extent necessary to permit their operators to be isolated. When pseudo-ops constitute the sole contents of a macro, they are taken as applying to the rest of the program in which they appear; the effect of calling on such a macro-ful of pseudo-ops is as if each pseudo-op were given as a separate input statement. Insofar as the notation a programmer requires is regular and self-consistent, then, it may be described in a single macro whose name might well be that of the language itself, and which would be called on at the beginning of any program written in that language. Statement forms that have special notational requirements in conflict with any global conventions would include the necessary local conventions within the bodies of their macro definitions. The local-notation feature will be illustrated in the next section.

As should be evident at this point, it is possible to teach XPOP to recognize an enormous number of logically identical but notationally different statements by means of a few uses of just those pseudo-ops introduced so far. It should be possible, in fact, to define a programming language empirically—that is, to treat a language as a cumulative, open-ended corpus of those statement forms that experience shows to be desirable. The full set of notation-defining pseudo-ops, of which about one-third is exhibited here, permits the description of the notations of FORTRAN, COBOL, and most other existing compiler languages.

Compilation-Control Pseudo-Operations

The compiler designer also needs, of course, various kinds of control over the compilation process. One requirement is for the ability to call for remote compilation. To meet this need XPOP provides the pseudo-ops WAIT and WAITIN. Both signify that the part of any macro lying within their range is to be expanded as usual—that is, parameters substituted for dummy variables, system-generated symbols inserted where called for, and so on—but that the resulting coding is not to be inserted into the object program yet. Instead, these instructions are put aside, to be inserted into the object program only when a source-program statement is found bearing the statement label specified by the WAIT or WAITIN. (The label to wait for is specified in the pseudo-op's variable field, where it may be given as a literal constant or—more likely—represented by a dummy to be replaced by a parameter.) In any case, all instructions waiting for such a label will appear just after those resulting from the translation of the statement so labeled.

The instructions waiting for a label may have come originally from several various macros, or several uses of the same macro; if so, the one difference between WAIT and WAITIN will make itself felt. If, for example, a group of instructions lay within range of WAIT ALPHA, they would be appended to the threaded list of those already waiting for ALPHA; if the pseudo-op were WAITIN, they would be prefixed to it. Those groups of instructions made to wait by WAITIN's will, therefore, appear in the object program in the inverse of the order in which they occurred in the source program—hence "WAITIN" (WAIT INverse). If the label for which a batch of instructions is waiting never appears, the instructions do not appear in the object program. If no label is specified, they appear at the very end of the object program.

The following example shows the use of WAITIN in a simplified version of FORTRAN's "DO"—one that permits only the special case of subscripting that is formally identical to indexing. First, the source program that defines "DO" to XPOP, and then uses it in a two-level-deep DO nest:

* Note that XPOP can process algebraic expressions. These may be used as source-language statements or within macros; when used within macros, they may contain dummy variables to be replaced by parameters when the macros are used, and those parameters may be arbitrarily long subexpressions. Subscripts, not now allowed, are being provided for.
J EQU 2
K EQU 4
CHPUNC 4S1=1 1,2, 1T2
DO MACRO A,B,C,D,01
)A AXT C,B
WAITIN A
TXI *+1,B,01
TXL )A+1,B,D
END
DO 15 J=1,3
DO 15 K=2,20,2
PHI,J=RHO,J+BETA,J
TAU,K=PHI,J+4
END

And below, the object program produced by the above:

J EQU 2
K EQU 4

CHPUNC 4S1 1=1,2, 1T2

NEW PARAMETER-STRING PUNCTUATION ADOPTED AT THIS POINT

DO MACRO A,B,C,D,01
)A AXT C,B
WAITIN A
TXI *+1,B,01
TXL )A+1,B,D
END

DO 15 J=1,3
DO 15 K=2,20,2
PHI,J=RHO,J+BETA,J
CLA BETA,J
FAD RHO,J
STO PHI,J
TAU,K=PHI,J+4
CLA =4
FAD PHI,J
STO TAU,K
TXI *+1,K,2
TXL )0002+1,K,20
TXI *+1,J,01
TXL )0001+1,J,3
END
Another obvious use for WAIT or WAITIN is the handling of closed subroutines. The programmer will frequently want a macro to generate only a calling sequence to a closed subroutine, with the subroutine itself appearing only once in the object program, at the end. To secure this effect, the programmer would define the macro in question as starting with the calling sequence; then he would incorporate a WAIT with blank variable field, a ONCE pseudo-op, and then the subroutine. If the macro were not used in a given source program, the subroutine would not be made part of the object program. If used, the first such use would output the calling sequence normally, and the subroutine as waiting instructions to be put into the object program at its end. Subsequent uses of the macro in that program would cause the compilation of the calling sequence only, the ONCE pseudo-op reminding XPOP that it had already compiled the subroutine. The following examples will illustrate uses of WAITIN, ONCE and local notation-defining pseudo-ops. The first is the pseudo-DO with its punctuation defined within its own body:

```
$DO  MACRO   A,B,C,D,01
$   CHPUNC  4S1  1=1,2,  1T2
$)A AXT     C,B
$   WAITIN  A
$   TXI     *+1,B,01
$   TXL     )A+1,B,D
$   END
$
$ DO 15 J=1,3
$   CHPUNC 4S1 1=1,2, 1T2
$NEW  PARAMETER-STRING PUNCTUATION ADOPTED AT THIS POINT
 )0001 AXT 1,J
$
$ DO 15 K=4,48,TWO
 )0002 AXT 4,K
$ PHIJ=RHO,J+BETA,J
 CLA   BETA,J
 FAD   RHO,J
 STO   PHIJ
 tau,K=PHIJ+4.
 15 CLA   =4.
 FAD   PHIJ
 STO   TAU,K
 TXI   *+1,K,TWO
 TXL   )0002+1,K,48
 TXI   *+1,J,01
 TXL   )0001+1,J,3
END
```

A use of ONCE is shown next. ONCE may be used in either of two ways, depending upon whether its variable field is blank or not. When the macro in which it occurs is being expanded and a ONCE with blank variable field is encountered, the name of the macro is searched for in a table. If it is found, the rest of that macro is skipped; if not, it is entered in the table to be found on later searches and expansion continues. The procedure followed if a symbol is found in the variable field differs only in that the symbol found is used rather than the name of the macro being expanded. This type of use permits copies of a subroutine, a set of constants, or a storage reservation to be incorporated into the definitions of many
different macros, with assurance that they will appear in the object program if and only if one of the macros is used, and not more than once no matter how many of them are used. It is this second type of use that is now shown:

\[ \text{FIRST MACRO A,B,C} \]
\[ \text{CLA A} \]
\[ \text{ADD B} \]
\[ \text{ONCE M} \]
\[ \text{STO C} \]
\[ \text{END} \]

\[ \text{SECOND MACRO X,Y,C} \]
\[ \text{LDQ X} \]
\[ \text{MPY Y} \]
\[ \text{ONCE M} \]
\[ \text{STO C} \]
\[ \text{END} \]

XECUTE Mode—A Compile-Time Execution Facility

The XPOP processor may at any point in a source program be switched into XECUTE mode, in which succeeding source-language
statements are not only compiled but assembled and executed. The programmer switches into this mode by using the pseudo-op \textsc{Xecute}, and reverts to normal processing by using the pseudo-op \textsc{Compyl}; the coding between each such pair is assembled as a batch, then executed. \textsc{Xecute} mode may be used with great freedom. The programmer may enter and depart it within a macro; while in the mode he may use macros (with full notational flexibility), algebraic expressions, and everything else that \textsc{Xpop} normally processes except certain pseudo-ops that would be meaningless at compile time. \textsc{Xecute} mode was originally implemented by those working on the \textsc{Xpop} processor for their own use in maintaining and developing that program, and has proved itself better for such tasks than any other method we know. It enables us to patch \textsc{Xpop} in a symbolic language practically identical to the \textsc{Fap} language in which the processor itself is written, and to cause these patches to become effective at such points during a compilation as we choose—not necessarily at load time. The effectiveness of any such patch can be made contingent on results of program execution thus far, so that tests otherwise requiring several machine runs can be accomplished in one. A \textsc{Fap}-like assembly listing is produced by \textsc{Xpop} while in \textsc{Xecute} mode, and the symbolic language employed is so nearly identical to \textsc{Fap} that the very cards used for \textsc{Xecute}-mode patches can later be used for \textsc{Fap} assembly-updating.*

But this facility is by no means usable only by those working on the processor itself. It has the further role of giving the compiler-designer working with \textsc{Xpop} the ability to specify pseudo-ops for his compiler, and make it perform any compile-time functions it requires that are not built into \textsc{Xpop}—building special tables, setting flags, and so on. It enables the designer to make his system, to any extent he wishes, an interpreter rather than a compiler, or a monitor/operating system rather than a language processor.

Compile-time execution makes a great variety of special effects readily available to the programmer. For example, it allows any macro to be used recursively: just before calling on itself, such a macro switches into \textsc{Xecute} mode, makes whatever test is required to determine whether further recursion is indicated, then switches back to compile normally either at or just after the internal call, depending on the outcome of that test. Another useful facility it affords is that of trapping any source-language statement type for such purposes as counting the number of uses made of it, taking snapshots of its variables before their values are changed, or debugging by testing the values of a procedure’s variables just before exiting from it. Such trapping could be done even at the machine-language level. If the programmer wanted to trap all \textsc{Tra} instructions, for example, he would define \textsc{Tra} to be a macro, enter \textsc{Xecute} mode within that macro to take the desired compile-time action, then return to normal processing. (The psuedo-op \textsc{Ultlev}—\textsc{Ultimate LeVeL} of expansion—would be used within such an op-code/macro to prevent the taking of a \textsc{Tra} instruction within the \textsc{Tra} macro as a recursive call, with resulting infinite regress.)

One purpose of replacing op codes by macros of the same name might be to cause each such extended operator to step a programmed clock at execution time (as well as executing the original op code, of course), so that the programmer can learn exactly how long his routines take to run—a critically important matter in real-time applications, which require that programmed procedures fit into time slots of fixed size. This capability, together with its notational-flexibility and immediate-execution features, makes \textsc{Xpop} particularly suitable for command and control programming.*

* We have produced a subroutine, entirely independent of \textsc{Xpop}, equivalent to \textsc{Xecute} mode, and hope soon to announce its general availability.

\textit{Language-Preserving Pseudo-Operations}

\textsc{Xpop} provides the programmer with a group of three pseudo-ops that enable him to order, at any points in his program, that all macros so far defined be punched onto binary cards, written onto tape, or both. The use of any of these pseudo-ops preserves all macros then in the system in a highly compact form (binary-card representation takes about one-sixth the number of cards that symbolic takes) and, more important, a form that can be read
into the system at tape speed on any later XPOP run, without the time-consuming process of scanning and compressing the symbolic-language definitions. Notation-defining macros may, of course, be preserved on cards and/or tape along with code-generating macros. The tape and/or card deck produced may thus contain a complete programming language of the programmer's own design in both vocabulary and notation. This language may then be changed in any respect during the course of any ordinary production or debugging run. Functions may be added or deleted, notation elaborated or simplified. Because any of these pseudo-ops can be used as often as desired in a single program, it is possible to preserve successively larger sets of macros, each set containing its predecessors as subsets, as well as any macros defined since. Each time macros are punched or written out by means of any of these pseudo-ops, a report is generated, giving an alphabetized list of the macros preserved and the percentage of the system's macro capacity they occupy.

Another two pseudo-ops are available for ordering, either during a later XPOP run, or later in the same run, that predefined macros be read in either from the input tape (if they had been preserved on cards) or a reserved tape (if they had been preserved on tape). Sets of preserved macros may be read into the system at any point in any program, making it possible to switch languages in midprogram. This greatly facilitates the consolidation into one program of sections written by several programmers using different XPOP-based languages—each section simply begins by reading into the system the language in which it was written.

The five pseudo-ops, and their exact effects, are given in Table 1.

As is shown in the following example, the programmer may override XPOP's built-in assumptions about the tapes that WMDT, WAPMD, and RMDT refer to. He does so simply by specifying, either by logical or by FORTRAN tape designation, the unit he wishes to address. He may also assign a name to each file when he creates it, and later retrieve it by name; this permits many languages to be stacked on a tape while sparing the programmer any concern over the position of the one of interest to him. In the example below, the programmer has used WMDT to write his language onto logical tape A6 under the name 'TEST'. His language consists of three macros, whose names are then listed by XPOP. (Since the amount of available core storage used by these three was less than one-half percent, it is given as zero percent.) He then read this language back in again, this time addressing the tape by its FORTRAN designation, 11.

```
$ WMDT TEST,A6
$THE FOLLOWING MACROS HAVE BEEN OUTPUT ON TAPE
$ TEST2
$ TESTER
$ TESTXC
$ 00 PER CENT OF AVAILABLE SPACE HAS BEEN USED
$ RMDT TEST,11
$ALL PREVIOUS MACROS HAVE BEEN DESTROYED BY THE USE OF RMDT
```

*Debugging Tools—The XRAY*

XPOP provides one unconventional tool for finding bugs that our experience has shown to be highly useful, and which might readily be incorporated into other systems. This is the XRAY—a structured, interpreted, and captioned dump of core memory and the output tape. It prints out the chief buffers, tables, and character-strings in the system in meaningful format and (where one exists) external representation, as well as all the program compiled so far (whether still in core or already on tape), and a standard octal dump of as much of core memory as the programmer may require. In case of system trouble or source-program trouble not covered by one of XPOP's 50-odd error messages, the first thing the XPOP programmer will want to check is that the macro definitions were properly accepted and packed away, and these definitions are accordingly converted back to original input form and exhibited first. Because these defini-
tions, as seen in an XRAY, have undergone both compression into internal form and expansion back into input form, the programmer who can recognize his macros there can feel some assurance that they were properly digested by XPOP. He will next want to see how the system has scanned the last statement it saw; for this purpose he is given a print-out of the table that shows what symbols XPOP extracted from that statement as the parameters, and how it paired them off with dummy variables. Following this he is shown that part of the compiled program still in the system's output buffer, then that part already written out onto tape. Finally, the XRAY will present as much of core memory in standard octal dump format as the programmer may have specified in the variable field of the XRAY pseudo-op that triggers this output.

XRAYs can be obtained in two ways. One is to use the pseudo-op explicitly at whatever points trouble has shown up in a previous run, or is to be feared; the other is to order compilation in XPER (eXPERimental) mode, which may be started at any point in the program by use of the pseudo-op XPER. In this mode, the detection by XPOP of any error in the source program or the system itself causes the generation of an XRAY—and one will be generated at the end of the program in any case. The two methods may be combined, the programmer calling explicitly for XRAYs at some points as well as compiling all or parts of his program in XPER mode. The information presented by an XRAY as presently constituted is not fully adequate (hence the selective octal dump as a backup), and additions to it are being made, but experience indicates that the gain in intelligibility of information presented in XRAY form over that given in octal dumps is great enough to mark a step forward in bug-finding methods, as we think that XEXECUTE mode does in bug-correction. The joint power of the two source language facilities suggests the possibility of some experiments in on-line debugging; we hope to report on these later.

ACKNOWLEDGMENTS

The general macro-instruction concept, as well as many details of format, are derived from the BE-SYS systems created at Bell Telephone Laboratories and generally associated with the names D. E. Eastwood and M. D. McIlroy.

Three projects similar at least in spirit to XPOP but which came to the author's attention too late to play any part in the XPOP design are the Generalized Assembly System (GAS) of G. H. Mealy, the Self-Extending Translator (SET) of R. K. Bennett and A. H. Kvilekval, and the Meta-Assembly Language of (presumably) D. E. Ferguson.

At Lockheed Missiles & Space Company the author's principal debt is to B. D. Rudin, W. F.
Main, and C. E. Duncan for steady faith and support over long bleak stretches. Much of the coding of the processor and most of the daily problems fell to W. H. Mead, Marion Miller, M. Roger Stark, and A. D. Stiegler.

The author is grateful to C. J. Shaw of System Development Corporation for an acute critique of XPOP that has helped to improve the presentation and to pinpoint the areas in which further development is most needed.10

Thanks are also due to P. Z. Ingerman of Westinghouse Electric Corporation for useful discussions on the relationship between XPOP and syntax-driven compilers, and for the opportunity to read part of his forthcoming book on such compilers.3

REFERENCES