

DESIGN AND IMPLEMENTATION OF A GENERAL
PURPOSE MACROPROCESSOR FOR
SOFTWARE CONVERSION

by

David A. Schmidt

B. A., Fort Hays Kansas State College, Hays, Kansas, 1975

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

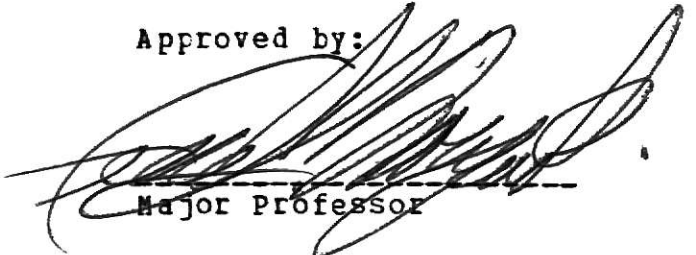
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:


Major Professor

Document

LD

2668

R4

1977

S34

C.2

i

ACKNOWLEDGEMENTS

Special thanks go to Gary Anderson, Dr. Fred Maryanski, and Rhonda Terry. This work was sponsored in part by U.S. Army research contract DAHC26-77-C-0003.

ILLEGIBLE DOCUMENT

**THE FOLLOWING
DOCUMENT(S) IS OF
POOR LEGIBILITY IN
THE ORIGINAL**

**THIS IS THE BEST
COPY AVAILABLE**

TABLE OF CONTENTS

1.0	Introduction.	1
2.0	Background.	4
2.1	Classification of Macros.	7
2.2	Examination of Existing Software for Possible Use.	19
3.0	Implementation of the Preprocessor.	28
3.1	Phases of Implementation.	29
4.0	Results of the Implementation	34
4.1	Application of the Preprocessor	35
4.2	Examples of Text Transformation	36
4.3	Hardware Requirements	40
5.0	Formal Definition of the Macro Language	44
5.1	Primitive Types	47
5.2	Symbols	47
5.3	Patterns.	48
5.4	Sets.	49
5.5	Transformation Classes.	50
5.6	Transformation Entries.	51
5.7	Classifications of Transforms and Priority Assignments.	52
6.0	Evaluation and Conclusions.	58
6.1	Extensibility and Future Uses	61

REFERENCES AND BIBLIOGRAPHY	65
---------------------------------------	----

APPENDICES

Breakdown of Construction Effort by Phases.	69
Preprocessor Input for Interdata COBOL Conversion.	71
Sample Preprocessor Output.	79

LIST OF FIGURES

Translation of Macros	6
A ML/I Macro Definition	25
Sample Proteus Input to Define and Use Complex Numbers	26
Sample Proteus Program Using Arithmetic Expressions.	27
Examples of Priority BNF Constructs	45
Sample Transformation Table	46

1.0 Introduction

One of the specific problems manifest in the overall area vaguely titled the "software crisis" of computer science is the maintenance of existing software. This maintenance can vary from the typical "cut and paste" modifications found with any active software all the way to a complete rewrite of a system's code. The latter is often the case when a user must change hardware or software support. Such conversion of existing programs from one language to another may not be a difficult task, but it is a time consuming one. To accomplish such a conversion manually usually produces a large amount of effort, cost, and errors. Automation of this task is therefore both desirable and challenging.

Automation of software conversion is not a new topic; the problems of machine and compiler incompatibilities have been present long enough to bring forth extensive work with macroprocessors and extensible languages. Consequently, the need to convert approximately 100,000 lines of IBM ANS COBOL to a version of ANS subset COBOL as implemented for the Interdata 8/32 suggested immediately the use of automation. Aside from the sheer volume of code needed to be converted, other reasons presented themselves as supporting points for design of an automatic conversion device:

-- the majority of changes involved were simple but time consuming, a situation which induces easy-to-make but hard-to-locate errors.

-- the lack of people experienced in the host and target languages meant that even the simplest of changes in the code would present formidable challenges to the personnel involved;

-- the probability of additional conversion in the near future of COBOL-coded data base management systems made any sort of automatic aid especially attractive.

Consequently, the decision was made to construct a device (denoted as a preprocessor) to automatically convert as many of the IBM to Interdata incompatibilities as possible. Since the target language in this situation was implemented via a minicomputer compiler, and since the area of minicomputer software is one undergoing a continual change in product, the preprocessor was to be designed to be as flexible as possible to meet these changes. That is, the device may be updated easily to handle new releases of the target language and new conversion situations as they may present themselves. One class of devices exhibiting such characteristics is said to be table driven, i.e., implemented is a "table" upon which the user states the target conversions and host replacements which are needed for the particular application. With such a method, only the table need be changed each time the device's application

is changed. (An example of a table-driven device is the so-called "compiler-compiler", a software tool designed for users desiring to create their own computer language with minimal effort.)

The automatic device to be used in the COBOL conversion was to be a table driven preprocessor. Since much work had already been accomplished in the area of programming language conversion, both in the situation previously mentioned and in the extension of programming languages, the next step in determining the design, implementation, and evaluation of a preprocessor was a survey of existing literature in the area.

2.0 Background

Literature in the field of programming languages suggested the use of a device called a macroprocessor for the COBOL conversion. A macroprocessor is "a piece of software which is designed to allow the user to add new facilities of his own design to an existing piece of software" [1]. More generally speaking, a macroprocessor supports the use of a software entity called a macro, which is nothing more than a symbol or sequence of symbols which are to be recognized and replaced with another, different sequence of symbols. How the macro is defined by the user and recognized by the macroprocessor is dependent upon the macroprocessor itself. Typically, macros are first defined by the use of a declaration mechanism similar to that used for declaring storage for program identifiers. The definition includes the calling format for the macro, by which recognition of the macro is later made, and the procedure for replacement of the macro by the expanded source text. Macros are used with an existing base language, and they are included in-line with program code. The macroprocessor then scans the input source code, recognizes the macro calls, and effects some sort of replacement. Macro calls typically have a parameter list, which is a segment of in-line source text needed to generate the desired expanded source. The resulting output is code now completely in the desired base language which can be now

successfully processed by a translator for the language. It should be noted that the idiom of a "host language" corresponds to the base language with the included macro calls, while the "target language" is the pure base language output. Figure 1 depicts this translation process.

FIGURE 1
TRANSLATION OF MACROS

HOST LANGUAGE
(BASE LANGUAGE WITH
IN-LINE MACRO CALLS)



MACROPROCESSOR
(CONVERTS CALLS TO
BASE LANGUAGE CODE)



EXPANDED SOURCE
(TARGET LANGUAGE)



TRANSLATOR FOR
TARGET LANGUAGE



OUTPUT OBJECT
CODE

When the macroprocessor is built into the target language's translator, the result is called an extensible language. This is because the base language may be "extended" upon the whim of the user to become useful in whatever application is desired. At the opposite end, when the macroprocessor is completely divorced from the target language's translator to the extent that the entire code body is first processed through the macroprocessor and then through the translator, the device is entitled a preprocessor. Macroprocessors may be either general purpose or special usage. A general purpose device may be tailored to accept a wide range of host and target languages, while a special usage device may not.

2.1 Classification of Macros

Macros may be classified by the means in which they are evaluated. Cheatham [2] lists three classes of macros:

- text macros: are evaluated by performing a lexical analysis (scan) upon the host language text;
- syntactic macros: are evaluated by performing a syntactic analysis (parse) upon the host language text;
- computational macros: are evaluated by performing analysis upon an intermediate code derived from the

host language source text.

The use of computational macros was not considered for the task at hand, as little use of such macros was found in language conversion tasks. Computational macros are more commonly implemented within extensible language translators. Of the two remaining classes, the syntactic macro is far more useful because it is capable of recognizing context sensitive macro calls where the text macro is not. Typically the text macro is also set off by some special keyword or delimiter which makes its use in applications other than a narrow range of language extensibility limited. Leavenworth [3] further defines the class of syntactic macros by creating two types: a "procedure oriented", text inserting macro called an SMACRO, and a "value returning" (function) macro called an FMACRO. The subclass of macro called the SMACRO is the focus of the literature survey.

Of prime importance when implementing and using macro definitions and macro calls is the inherent capabilities of the macro as supported by its processor. McIlroy [4] suggests a list for evaluating a macro's capabilities. Briefly stated, the "ideal" macro should support:

- pyramided calls: the nesting of a macro call within a macro call, i.e., the text generated by a macro can contain additional macro calls which are evaluated as if they were present in the original source;

-- conditional calls: the substitution of the expanded text can be made dependent upon program conditions previously defined or upon the parameters passed with the macro call.

-- creation of source text symbols: the use within the expanded program source of identifiers and labels generated by the macroprocessor so to completely effect the transformation;

-- grouping of parameter values: the use of a mechanism (such as parentheses) to allow the passing of a list of parameters in such a fashion so to establish explicit precedences upon the parameters' evaluation. (These precedences can be compared to the precedences established by the use of parentheses with arithmetic operators in numeric expression evaluation.) This allows the passing of variable length lists of parameters (i.e., program text symbols) which can be correctly interpreted by the macroprocessor for translation.

-- nested definition: the ability to establish a new macro declaration (definition) by including such in the expanded source text inserted by the evaluator of the current macro. The declaration can then be processed as if it was previously present in the original source text.

-- macro repetition (recursion): the ability of the macro to recall itself dependent upon the parameter

values of the macro call (and so the text generated).

Whether or not these macro capabilities can be realized is a function of the way in which the macroprocessor is designed and implemented. Brown [5] gives an excellent checklist in this regard; a summary of it follows. Basically five items must be taken into consideration when designing a macroprocessor: the base language (target language) to be used, the syntax of the macro calls, the means used in macro evaluation, the macro-time facilities available, and the methods of implementation used.

The choice of a base language for the macroprocessor is a major one; a general purpose device is designed so that the user may apply the macroprocessor with any base language desired. This generality usually produces device complexity and limits the transformation powers. The special purpose macroprocessor is limited in its range of applicability, but often its power is enhanced by the knowledge of the format and syntax of the output. The macroprocessor is designed around the target language. Macroprocessor use has been most prevalent in specialized applications, although this may be more of a function of the disposition of the knowledgeable user to turn to macroprocessors in such situations, as opposed to the general purpose user, who often acquires a new translator instead.

The syntax associated with a macro call often determines how the macro is capable of being evaluated. The

macro call may be recognized by the processor in a number of ways. The use of name recognition ("keywords" signalling macro evaluation, much like a FORTRAN subroutine call) is simple to implement but limited in use; it restricts evaluation effectively at the lexical analysis level. Syntactic evaluation of the call is usually accomplished by some means of pattern matching. This scheme usually means the source text must be tokenized (i.e., the varying length character strings are converted to an internal representation where one symbol corresponds to each string), which realizes extra overhead upon the device. Pattern matching can be effected upon non-tokenized text, but the task of matching a character at a time is so time consuming that severe restrictions must be placed upon the calling format. Often formal delimiters such as '\$', or end-of-line characters, must be used.

Accompanying the macro call must be the macro's parameter list. This list can range from a formal specification of identifiers enclosed in parentheses and separated by commas to a variable length, format-free listing which is indistinguishable from the rest of the base language code. Again a trade-off exists between ease of evaluation and power of usage. Ideally, the macroprocessor should be able to accept a list like the latter and treat it with the overhead usually attributed to the former. Another consideration in parameter list evaluation is the way in which the macroprocessor accepts the parameters as per the

macro definition. Parameters may be accepted by number (e.g., a list of n parameters where each entry can be referenced by its displacement within the list), or by name, where recognition is dependent not upon ordering, but by the parameters' text representation.

The situations in which the macro call may be recognized is also a consideration in macro syntax. The notation independent macro can be identified irregardless of its position in the source text, and without use of special delimiters. This allows the macro calls to fit into the base language naturally and promotes an ease of usage. A good extensible language provides such a feature. This method can be contrasted to the macro call which must be located in a special position in the text and set off by special delimiters. Further power can be given the macro call if its replacement text can be conditionally generated dependent upon program conditions of parameter attributes. This macro call negation allows the user to selectively activate and deactivate macro calls without rewriting the source text.

Text evaluation of the macro call may take on many forms. This area relates closely to McIlroy's checklist, as the means of evaluation is directly related to the macro's power. A desirable feature of macro evaluation is that it be recursive, i.e., the expanded replacement text may contain calls to the macro which instituted the replacement. The range of the macro call also improves its generality and

power: a call which can extend over several lines (i.e., a "multilevel call") is particularly useful for text recognition and optimization. The time at which the macro call's parameters are evaluated is another consideration. The parameters can be evaluated (i.e., expanded, if they contain macro calls) immediately before the macro call itself is replaced. Such a procedure is a call by value. A delayed evaluation until after the replacement text has been generated is a call by name. The two different forms of evaluation produce different results, and the call by name is considered more powerful as it facilitates ascendant and descendant macro calls [6].

A macro's scope in the source text must also be reckoned with in terms of text evaluation. A global macro is in force from the point in which it is declared (or even beforehand, in the case of recursive devices) until the end of the source text. A local macro can be "turned on and off" at will (an illustration of this concept is the use of the ACTIVATE and DEACTIVATE verbs in the PL/I preprocessor [7]). Global macros are often preferred because they introduce a consistency in evaluation which is violated by the concept of locality.

Generation of the replacement text for a macro call is a responsibility of the macro-time facilities of the processor. Two obvious considerations are the use of macro-time variables and created symbols. Macro-time variables are value holders used by the macro procedures for

facilitating text replacement. They correspond directly to the variable declaration found in any user program. Macro variables may either be local or global; local variables are active only when the macro evaluation routine is called, while global variables contain values which are accessible by all macro routines. Use of global variables allow the macro routines to communicate status to one another thus encouraging the use of conditional replacement and multilevel calls. In addition, global variables can be established as quite complex data structures which allow sophisticated evaluation and replacement.

Any macro routine that is considered to have text replacement power must be capable of generating its own created symbols to be inserted into the replacement text. This includes new labels as well as identifiers. The problem of the identifier's declaration is often brought about by this feature, and so the macroprocessor must be capable of accounting for the solution of this problem. Obviously any newly created symbols may not conflict with already existing ones; some means need be established to guarantee completely as possible that such a redundancy not occur.

The insertion of the replacement text into the source is directly determined by macro-time statements which are a part of the macro language. Some definitions are in order here for the sake of clarification. A macro definition usually consists of two parts: the macro calling syntax

definition (known as the "macro head") and the supplied instructions for creation and insertion of the replacement text (the "macro body"). The replacement instructions can be quite simple in form (e.g., no instructions at all--simply replacement text) or can utilize such constructs as arithmetic, loop control, and string manipulation statements (SNOBOL must be mentioned here as an example of such power [8]). McIlroy believes that these macro-time statements which form the macro body should be every bit as powerful as those found in an algebraic language [9]. The designers of the PL/I preprocessor took such advice to heart by allowing PL/I to be the macro language of their preprocessor. Unfortunately, macroprocessors need string manipulation capabilities more than complex arithmetic evaluation features, and so a close examination must be undertaken whenever implementing a macro-time language.

An extension of the use of global macro-time variables is the creation of a macro-time dictionary which is the macroprocessor equivalent of a compiler's symbol tables. Such a dictionary can be used to hold source text identifiers and their determined attributes. The system is a boon to complex conditional replacement, as a wealth of information may be extracted by one macro for later use by another. The dictionary can be further extended to contain "canned" macro routines for use as a systems library by the processor. Ease in user coding is thus promoted. The concept of a macro-time dictionary extends the

macroprocessor's power to the point that it can become ultimately a full-fledged compiler.

Once the design of the macroprocessor has been determined, the method of implementation is brought to the fore. As in all translators, a number of fundamental decisions concerning construction must be made. Is the device to be one pass or multi-pass? A multi-pass device is typically slow but does present the advantages of producing a device which can fit into a smaller main memory (due to overlaying) and can build a more complete macro dictionary (due to the extra scans obtained). A novel approach is to construct a one-pass device which is reentrant, i.e., after one pass of the source text, the processor reenters itself to perform additional scans. An advantage of such a device is that only one memory load is needed; size is obviously not reduced.

Storage of text information can be done internally with the use of contiguous lists, linked lists, or stacks. Stacks are more useful for temporary storage, contiguous lists for permanent storage, and linked lists for use of dynamically allocated, variable length storage. A decided advantage in list processing is the capabilities provided towards symbol manipulation. Special purpose devices have the advantage of tailoring the storage mechanisms to the language being processed (e.g., stack storage mechanisms are used in all ALGOL compilers) and thus can optimize memory sizes and execution speeds in this area. General purpose

devices often must estimate storage requirements and must allocate dynamically. Often all three methods are used in a single device.

The factor of execution speed is most strongly influenced by the method of macro recognition used. The use of keying words and delimiters promotes quickest evaluation at a loss of generality and power. The designer must decide whether the execution speed is an important enough consideration to sacrifice such generality. Often execution speeds can be improved when the macro processing is overlapped with I/O to backing secondary storage.

Since the ultimate objective of any macroprocessor is to be operationable, user considerations should be given prime weight in the device's design and implementation. Ideally the macroprocessor should support:

- transformations which allow it to be used as a powerful text editor. In this fashion, the operations of text editing and expansion can be combined into one.
- macro calls which blend in well with the base language. The user should be able to utilize the calls easily so as to forget that they are actually alien to the language itself: the macro calls thus become transparent.
- ease in coding macro definitions so that the definition of a macro need not be left solely in the hands of a few dedicated systems programmers. The

macro language should be easy to learn and use.

-- error detection and recovery of improperly coded macro calls. This is currently a difficult problem with most macroprocessors as typically an improperly coded macro call is not recognized at all and so passes by the control of the macroprocessor and is flagged by the base language translator. At best, the macro is improperly translated. This presents problems to the user as: 1) the translator error messages do not indicate that the macro was coded improperly; 2) the translator listing reflects the expanded output, and the original source macros have disappeared completely from the text. A user who is unaware of the macro translation process will find the reading of such output an impossible task. The macroprocessor should have some sort of error detection to at least mark statements which appear to be improperly coded and an output mechanism to present the original source along with the error listings.

The considerations mentioned here for macroprocessors also apply to those processors implemented as part of the base language translator to form the foundation of an extensible language. In particular, an extensible language is concerned with the transparency of usage of macros, and the macros' abilities to initiate new operators, data types, and language verbs. Ideally, an extensible language

presents a "core" of data types and statements upon which the user builds his own custom version of the language. Although a device to facilitate such a powerful expansion would be difficult to implement for a wide variety of languages, the processor in an extensible language is a special purpose device and so can take advantage of the advance knowledge such a situation entails. As with macroprocessors, the processor within the extensible language's translator may expand macro calls during the lexical, syntactic, or code analysis phases. Since both macroprocessors and extensible languages deal basically with the same problem, the two will be treated as one group.

2.2 Examination of Existing Software for Possible Use

A survey of the requirements of the COBCL conversion project versus the capabilities available in macroprocessors produced the following list of desired features for the soon-to-be constructed device:

- the device should be table driven so that macro definitions can be added, changed, and deleted easily and without altering the macroprocessor code itself;
- macro calls must be notation independent, as the conversions needed for implementation may occur at any

position in the source and are not set off by any special delimiter. The calls must be, in effect, transparent, as in actuality they are ANS COBOL-coded verbs which are not implemented by the Interdata compiler.

-- the macro definitions must be completely divorced from the source code for the reason specified above. The definitions must be allocated from a separate source.

-- the parameter list of the macro call must be able to handle variable length parameters or have some mechanism for "collecting" a list of parameters into a single entity. This feature is necessitated by the saving and transportatation of such items as variable length clauses and expressions.

-- the device can best fulfill the needs at hand if it is syntax-driven. This allows the handling of complex, variable length macro calls in a fashion which encourages subsequent text generation.

-- the ability to define and use new types (such as "literal" and "expression") is highly desirable.

-- the macro-time language must facilitate conditional replacement and table handling facilities. The language conversion task necessitates the building of auxiliary "symbol tables" to be used in the processing of later macro calls; often text substitution is conditional upon the results of an earlier macro call.

These problems are easily solved with the desired features.

-- the macroprocessor must be implementable on the Interdata 8/32 within a period of three months.

-- the macroprocessor must be coded in a language which is easy to read and modify.

The latter two points on this checklist are particularly important as a time deadline was in effect for implementing the device, but the result was to be general purpose enough so that later programmers could tailor the device to their own specific needs. In other words, the macroprocessor must first and foremost be easy to implement, easy to understand, and easy to use. To speed up the implementation process, a survey was made of the existing general purpose macroprocessors available which contained features needed for the conversion task. The examination of the field will not be reproduced here in full; much documentation already exists on these devices. Three existing processors met with more than casual interest; their features will be briefly listed below.

One macroprocessor which stood forward with a number of features matching the requirements was P. J. Brown's ML/I [10], a general purpose macroprocessor. ML/I is applicable to the situation because:

-- the user can define the format of the macro calls in

any fashion he so desires. This allows calls in the fashion of:

```
IF arg1 = arg2 THEN arg3 ELSE arg4.
```

where each `argi` is a variable length list of identifiers.

-- ML/I allows conditional generation of replacement text as well as use of macro-time variables and symbol creation.

Negative points concerning ML/I were found to be:

-- no type checking of parameters is accomplished by the macro call;

-- handling of variable length macro calls is awkward to code and understand as it involves an iterative mechanism (shown in Figure 2);

-- the application of nested macro calls is defined in a manner which is difficult to understand and may lead to incorrect results.

ML/I's lack of type checking proved to be a particular problem in evaluating variable length calls, as the user has to compensate by generating macro-time code which uses artificial "nodal points" in the calling format to allow an iteration on the parameter list. Figure 2 depicts such an ML/I macro definition which decodes an arbitrary length assignment statement into a sequence of assembly level

instructions.

An extensible language which was found to contain a large number of features desired in the final product was Proteus, a language designed and implemented by James Bell [11]. Proteus performed macro substitution with the use of pseudo-BNF rules called transformations. These transformations were especially attractive in that they resembled Backus-Naur Form closely, used explicit priority values to arrange the ordering of macro evaluation, and could be organized as a table and manipulated easily. Proteus also presented a strong position due to the following other points:

- macro calls are syntax-driven, notation independent, and totally transparent;
- the basic text replacement mechanism is inherent within the pseudo-BNF transformation statement;
- additional macro-time text manipulation can be invoked through the use of immediate evaluation ("action routines") or delayed evaluation ("semantic routines");
- creation and parsing of new types is simply accomplished;
- the processor has been successfully implemented using FORTRAN II, and a source code listing of the complete interpreter was available [12].

An example of Proteus in use is shown in Figure 3, where a type complex is defined to represent a complex number in terms of reals. An accompanying transformation shows how addition of complex numbers is interpreted. Note that immediate actions are followed by exclamation points; delayed actions are followed by semicolons. A priority number accompanies each transformation. A Proteus program to interpret the construct processed by ML/I in Figure 2 is shown in Figure 4.

The third available language considered for implementation was SNOBOL4. Besides being a general purpose, powerful, pattern matching language, SNOBOL was advantageous in that it allowed notation independent calls and rudimentary type assignment. Unfortunately, no version of SNOBOL was available for use on the Interdata 8/32, and use of the language implied the added task of implementation of SNOBOL on the 8/32 in the allotted period of time.

FIGURE 2

A ML/1 MACRO DEFINITION

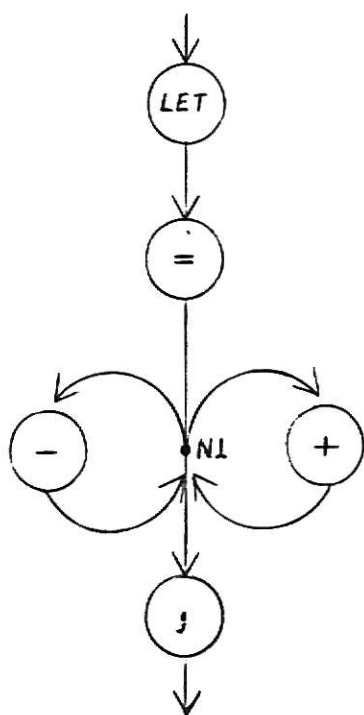
STATEMENT FORMAT:

LET identifier = identifier (+|-) identifier;

EXAMPLE:

LET A = B - C + D;

MACRO DEFINITION:



```

MCDEF
LET = N1 OPT + N1 OR - N1 OR ;
  ALL
  < LOAD %A2.
  MCSET T1 = 3;
  %L4. MCGO L2 IF %DT1 - 1 = +;
        MCGO L5 UNLESS %TT1 - 1 = -;
        SUB %AT1
        MCGO L3
  %L2. ADD %AT1.
  %L3. MCSET T1 = T1 + 1;
        MCGO L4;
  %L5. STCRE %A1    >;
  
```

PRODUCES: LOAD B
 SUB C
 ADD D
 STORE A

FIGURE 3

SAMPLE PROTEUS INPUT TO DEFINE AND USE COMPLEX NUMBERS

THE TEXT:

```

PATTERN COMPLEX| COMPLEX <- "<REAL: real> + <IMAG: real> I" |
trans 30 "<X: complex> + <Y: complex>" <- "<Z:complex>" |
      Z.REAL <- X.REAL + Y.REAL,
      Z.IMAG <- X.IMAG + Y.IMAG;

```

ALLOWS THE USE OF:

$$4 + 3I$$

$$6 + 2.5I$$

AS WELL AS:

$$4 + 3I + 6 + 2.5I$$

TO PRODUCE:

$$10 + 5.5I$$

FIGURE 4
SAMPLE PROTEUS PROGRAM USING ARITHMETIC EXPRESSIONS

```

PATTERN ID| ID <- "<INT: int>"|
trans 40 "<A: id> + <B: id>" <- "<C: id>"|
      C.INT <- A.INT + B.INT;
trans 40 "<A: id> - <B: id>" <- "<C: id>"|
      C.INT <- A.INT - B.INT;
trans 30 "LET <A: id> = <B: id>" <- ""|
      A.INT <- B.INT;

ID A, B, C, D|
.
.
.
LET A = B - C + D|

```

3.0 Implementation of the Preprocessor

Bell's Proteus language presented the best position from which to proceed towards a workable macroprocessor. The decision was made to extract from Bell's interpreter the pattern matching and text insertion mechanisms of the code and use them as the basis of the new device. Since the end result was to be a preprocessor, Bell's semantic routines were not needed and so were discarded. A problem was encountered in preparing for modification as it was found that Bell's FORTRAN implementation was difficult to read and not clearly modularized. As a result, it was decided to convert the extracted FORTRAN code to sequential PASCAL [13] and streamline it to be both modular and readable. Use of sequential PASCAL over FORTRAN provided additional advantages as well. With PASCAL the capabilities existed for set handling, dynamic allocation of storage, and block nesting, all of which would prove to be definite aids in device construction. In addition, PASCAL provides a character type primitive which allows for simpler text I/O. Disadvantages inherent in using PASCAL were rooted in the current implementation, which used an interpretive mechanism in an emulation of the DEC PDP-11. Once the basic tokenization, pattern matching, and text insertion routines were made operational, a step-by-step modification would be made on the device to produce a general purpose preprocessor geared toward COBOL to COBOL conversion.

3.1 Phases of Implementation

Complete implementation of the preprocessor can be viewed as an evolution spanning four phases. Each phase will be examined separately in the following paragraphs. As a brief overview, the phases are:

1. The conversion of the FORTRAN II version of the Proteus interpreter to an equivalent PASCAL version;
2. The addition of formatting and input-output routines applicable for the COBOL conversion task;
3. The conversion of the program into a two-pass device;
4. The creation of the macros needed for the COBOL conversion.

The first phase of implementation was concerned with carrying over the pattern matching and text transformations of Bell's interpreter to a working PASCAL replica. Consideration of the macro call syntax was in order. Bell's syntax was adhered to faithfully with the following exceptions:

- the left (replacement) and right (matched) pattern strings were clarified and redesigned to correspond more closely to EBNF notation;
- the presence of the action routine statements immediately following the left and right pattern

strings was dropped; the action routine statements were moved to a position internal to the preprocessor, to be invoked by a number supplied with the transformation called the action routine number.

-- the use of semantic routines (used by Bell as a code generation-interpretation device) was dropped.

During implementation, it was found that Bell's mechanism for tokenizing text used an inefficient tree structure to store the original text. This portion of code was dropped and replaced by the tokenizing mechanism used in the concurrent PASCAL compiler implemented by Hartmann [14]. The latter version was faster and simpler to implement and alter.

Bell's pattern matching and text replacement routines were converted faithfully; their simple mechanisms proved easy to use and understand. Almost all of the data structures created by Bell for the device were implemented as given. This includes assorted text and token buffers, the transformation table and its associated pattern list, and a symbol table used for holding type classifications of tokens. Deleted was a "memory" array used to simulate dynamic storage allocation, as actual allocation was available with PASCAL.

Conversion in phase one progressed smoothly, although hardware problems were a major factor in delays. Bell's routines worked correctly in PASCAL, and the readability

factor of the new language was a decided plus.

The second phase of the preprocessor implementation involved coding of procedures designed to accept the IBM COBOL formatted input and convert it for correct tokenization (Bell's device relied on blank-sensitive, format-free input). Problems were encountered in dealing with the flexibility COBOL provides in continuing identifiers and literal strings onto new lines. Routines were written to correctly reassemble such continuations. Literal strings were also extracted from the source and replaced by special markers; comments were extracted likewise.

One major task handled in phase two was the necessity of saving all formatting and spacing information inherent in the original source so as to produce an output which is formatted identically to the input. Since Bell's device was a translator unconcerned with such a problem, routines were created from scratch to insert in-line within the tokenized text, tokens which contained spacing information. These formatting tokens were established in such a manner so as to be transparent to the pattern matching and text insertion processes. Upon output, the tokens are decoded and the original formatting is restored.

Phase three of preprocessor construction was necessitated by the realization that the code implemented to this point had filled available core to an unsafe state (considering that actual processing of text would require

significant amounts of data space). The device was divided into two passes to more effectively utilize code space. The resulting first pass input the transformation patterns, tokenized the input source text, and output the results to secondary storage. Pass two then performed the pattern matching, text insertion, and output of the expanded source. The two passes were monitored by a newly created driver routine, which acted interactively with the user console to provide extended capabilities in listing, trace, and error message transmission from the device. A large number of new procedures had to be coded to facilitate the major conversion, and slowdown due to lack of knowledge concerning PASCAL's file handling conventions presented problems.

The final phase of the preprocessor implementation concerned itself with the actual construction of a user transformation table for COBOL to COBOL conversion and the coding of necessary action routines. As implementation of the transformations progressed, it became apparent that Bell's version of pattern matching was not optimal for the type of text transformation desired. Consequently, the semantics of his BNF notation were expanded to include a wider range of possible combination of symbols within patterns, and the basic pattern matching mechanism was rewritten to accommodate this change. Certain ambiguities concerning the use of labeled nonterminals in Bell's matching scheme were also noted and clarified. The result is a precisely defined mechanism which is formally defined

for the input grammar in the following chapter.

Coding of action routines to aid in in-line text insertion and deletion and out-of-line text generation was performed near the end of this phase. Since the macro-time language used was PASCAL itself, unusually effective routines could be generated to access all the device's global tables and create data structures of their own. The routines can easily signal one another and effect virtually any sort of text transformation needed. The results of these efforts are reflected in Chapter 5 and in the preprocessor's users manual [15].

Construction of the basic device was considered complete at this point. Work still continues, however, as the preprocessor is put to actual use in COBOL conversion. Thus far, no major flaws have been encountered, and virtually all the major objectives set forth in the previous chapter seem to have been met.

4.0 Results of the Implementation

Initial use of the developed preprocessor (named "PRECOB") has brought forth three main conclusions:

1) The device is usable at several levels. A casual user of the preprocessor need know nothing about the device's operation if a macro table (transformation) has been already prepared. All the skill needed is the submission of a one line command from the user console giving device name, source and destination files, and list and trace options. A macro deck is also input to the card reader. A user who wishes to add new macros for the specific run can learn the syntax of the macro calls (priority BNF) in a short amount of time and add transformations to the macro deck quickly and easily. A serious user of the device is provided ample documentation via the device users manual and can generate action routines to effect powerful, comprehensive text transformation. These different levels of use of the preprocessor allow it to interact with a wide range of users with good results.

2) The device is portable and easily modifiable. The coding of the preprocessor in sequential PASCAL provides an easy to read, well documented representation of the macro language semantics. Conversion to some other language would present no major problems. The accompanying user documentation

provides a comprehensive explanation of every major module in the device. Modification to the preprocessor is facilitated by the modular breakdown of the device's functions. Procedures can be easily inserted and deleted, and numerous small modifications to the original code source have attested to the sound layout of the program's modules.

3) The device is flexible enough to function as a general purpose macroprocessor. During construction, the functions of the preprocessor were kept at a base language independent level which allows for simple modification whenever a new base language is instituted. All routines added for facilitating proper formatting and parsing of the input and output were separated as completely as possible from the device framework and clearly labeled as language independent. Use of the device in some new function means the easy removal or modification of this code.

4.1 Application of the Preprocessor

As stated in the introduction to this report, the motivation for construction of the preprocessor was the necessity of converting large amounts of IBM ANS COBOL text to a form which would compile and execute correctly on the Interdata 8/32 minicomputer using its ANS subset COBOL

compiler. The initial objective was to construct a transformation table with accompanying action routines to automate a minimum of 90% of the conversions (by number) necessary for the COBOL code. This goal has been met with a set of transformations and routines which

- standardizes COBOL text, eliminating "noise words";
- converts simple IBM-Interdata incompatibilities through the use of conditional and iterative text generation;
- converts major IBM-Interdata COBOL incompatibilities through the generation of out-of-line text;
- outputs as comments any original source which is the object of a major conversion.

Each group is examined separately with examples.

4.2 Examples of Text Transformation

Since many COBOL words are optional or have multiple spellings, some standardization of text is needed to allow consistent matching of expected patterns. Examples of such reductions are:

```

THRU    <- THROUGH
(null)  <- ,
        ;
        IS
        ARE

```

```

      AT
ZERO   <- ZEROES
      ZEROS
VALUE  <- VALUES

```

Note the elimination of the punctuation characters which can be used freely throughout COBOL text.

Simple conversions in the COBOL task involve those items which are subject to IBM COBOL "abbreviation" aids or are lacking in the current Interdata COBOL compiler. All simple conversions produce output which is consistent with the original source. Examples of these conversions (with critical portions underlined) are:

-- elimination of recording mode in file description clauses (not supported in Interdata COBOL)

```
FD AFILE RECORDING MODE FIXED
```

-- changing of label records clauses to OMITTED (not supported in Interdata COBOL)

```
FD AFILE LABEL RECORDS STANDARD
```

-- elimination of signed table indicies (implementation dependent)

```
SET INDEXA TO ± 1
```

-- elimination of variable length array usage (not supported in Interdata COBOL)

```
01 TABLEA.
  02 ITEM OCCURS 4 TO 10 TIMES
    DEPENDING ON X PIC 9.
```

-- elimination of 88 level mnemonic declaration and use (not supported in Interdata COBOL)

```
77 STUDENT PIC 9.
  88 GRADUATE VALUE 5.
    .
    .
    .
```

IF GRADUATE GC TO FINISH.

substituted for the latter statement would be

IF STUDENT = 5 ...

-- expansion of conditional expressions (IBM abbreviation)

IF A = 1 OR 2 OR 3 STOP RUN.

the expansion reads

IF A = 1 OR A = 2 OR A = 3 ...

Major IBM-Interdata conversions involve powerful COBOL verbs which have not been implemented in the Interdata compiler. The approach for conversion is to replace the nonavailable verb with a calling statement (COBOL PERFORM) to an appended routine (COBOL paragraph) which simulated the original text's actions. Two examples of major conversions are given.

-- the expansion of SEARCH and SEARCH ALL statements into PERFORMs which invoke generated out-of-line text. A sample SEARCH ALL is:

```
SEARCH ALL TABLEA
  AT END GC TO PARAE
  WHEN ITEMS(INDEXA) = 1
  MOVE 1 TO FLAG.
```

this text is expanded to

```
SET INDEXA TO 1
MOVE 0 TO FINISHED(01)
PERFORM SEARCH01 UNTIL FINISHED(01) = 1.
```

the invoked routine SEARCH01 is appended at the end of the code body and reads

```

SEARCH01.
  IF INDEXA > 10
    MOVE 1 TO FINISHED(01)
    GO TO PARAB
  ELSE IF ITEMS(INDEXA) = 1
    MCVE 1 TO FINISHED(01)
    MOVE 1 TO FLAG
  ELSE SET INDEXA UP BY 1.

```

The conversion of the SEARCH-SEARCH ALL requires the gathering of information from the source program's DATA DIVISION concerning table size and index usage. Automatic SEARCH conversion is a major feat.

-- expansion of PERFORM...VARYING statement to a PERFCRM...UNTIL with accompanying out-of-line code.

```

PERFORM PARA-a VARYING I FROM 2
  BY 1 UNTIL I = 10

```

becomes

```

MCVE 2 TC I
MOVE 0 TO FINISHED(02)
PERFORM PERFORM02 UNTIL FINISHED(02) = 1

```

with out-of-line text

```

PERFORM02.
  IF I =10
    THEN MOVE 1 TO FINISHED(02)
  ELSE PERFORM PARA-a
    ADD 1 TO I.

```

Time limitations have prevented the creation of a transformation table which approaches the 100% mark in conversion. As with all porting projects, incompatibilities exist which can not be automated due to serious hardware differences in device and file management. Nevertheless, the automation of those simpler text problems free the programmer to examine those difficulties which are worthy of his time and skill.

4.3 Hardware Requirements

One important factor in any large program written for minicomputer use is the demand the code puts on main memory and peripheral devices. Consequently, the memory and device needs of the PRECOB preprocessor are stated. Before examination of the data is made, one premise must be kept in mind: all figures are dependent upon the current implementation of PASCAL on the Interdata 8/32 at Kansas State University. This implementation is an emulation of the original PASCAL system designed by Per Brinch Hansen for the PDP-11/45 at the California Institute of Technology [16] and is most definitely not optimal for the Interdata architecture.

Execution speeds for the preprocessor vary due to several factors:

- the number of conversions to be searched for in the text;
- the size and complexity of conversions that are actually performed;
- the number of text words in the input source program;
- the size of the program sentences in the input source program.

Of the points listed, only the last entry needs further

explanation. The preprocessor works best with small program sentences; this is due to its simple, nonoptimized implementation. Since the program works on the input source one text sentence at a time, small sentences allow for quicker transformation and output. Tests made with "typical" COBOL programs in a "typical" environment (twelve words per COBOL sentence; a transformation table with fifty entries) show a processing rate of approximately 230 source text words per minute (which equates to about sixty lines of COBOL source code). The slow speeds are partly due to the device's simplistic and exhaustive pattern matching methods and partly due to the interpretive environment in which it must execute.

Core requirements for the preprocessor's object code is currently 18.5K bytes; the area used for table building and literal constants occupies 24K bytes. The latter figure is a "safe amount" in that table sizes vary from execution to execution and often some of this space is left unused. At most, the 24K bytes will allow the building of tables to facilitate the processing of input source with approximately one thousand identifiers. (It should be noted here that a newer version of the preprocessor has been constructed which reuses this space and additional secondary storage to allow programs with up to 4500 identifiers. Unfortunately, execution times are slower by a factor of three.)

Since all sequential PASCAL programs must run under the control of a concurrent PASCAL process, space must also be

allocated for the SOLO operating system. The version of SOLO currently in use occupies 8.5K bytes of object code space, and approximately 27K bytes is used for shared data space needed for reentrant code and kernel-interpreter interfaces.

The kernel and interpreter are assembly level programs which are used to interpret the PASCAL object code and interface with the OS-32MT operating system present on the Interdata 8/32. Together the two occupy an area of 7.5K bytes.

These parts are located in the 8/32 by placing the kernel-interpreter module in the Interdata's run-time library and the remaining parts into a partition of size 80K bytes.

The preprocessor utilizes a card reader, line printer, and disk drive during its execution. Use of secondary storage is particularly interesting because the PASCAL system utilizes a "virtual disk" which contains all the source and object code accessible to the running SOLO system. This virtual disk is currently implemented as an OS-32MT contiguous disk file of 9600, 256 byte sectors. This file is used by the PASCAL system to create sub-files which are the "files" of a PASCAL system. A single PASCAL file is limited to 129,560 bytes.

Use of the preprocessor in this environment has uncovered a number of potential problems which are solvable with varying amounts of effort. These difficulties are:

- the sizes of the two passes are still too large to permit a large amount of growth;
- the limited data space necessitates the storage of some tables in secondary instead of primary memory;
- the limit on the length of a PASCAL file prevents the current implementation from processing excessively large sequences of source text (this problem has been surmounted in the newer version previously mentioned).

Current work has been focused on the latter two points as the features necessary for the COBOL conversion project have been successfully installed in the device as it now exists. Reduction of pass size seems to be best accomplished by conversion to a four pass device; data tables can be easily transferred to disk storage; and input and output text can be stored on tape. Consequently, no major problems are foreseen for the constructed device. In addition, the preprocessor's options will be expanded considerably when an Interdata-based PASCAL system is implemented.

5.0 Formal Definition of the Macro Language

To this point, little has been said of the macro language used by the preprocessor. Ample examples of its syntax and use exist in the users manual [17], but no formal definition has been presented for priority BNF. In order to clarify any semantic questions and establish the functions of the macro language in relation to the preprocessor, a definition has been delineated (see Figures 5 and 6 for examples of the grammatical constructs to be defined). This definition encompasses the elements of priority BNF, their relation to the input source, the semantics of priority BNF transformations, and the interrelationships between the transformations of a transformation table.

It is important to note that the macro language given here is not the same as the notation utilized by James Bell in his Proteus language [18]. Most of the basic symbols and operations have been carried over, but the available functions of the macro language have been expanded, and the semantics of the operations have been changed.

FIGURE 5
EXAMPLES OF
PRIORITY BNF CONSTRUCTS

TERMINAL SYMBOL:

ARE

IF

XYZ

NONTERMINAL SYMBOL:

<DELIMITER>

<EXPRESSION>

LABELED NONTERMINAL SYMBOL:

<A:DELIM>

<Y:ANY>

PATTERN:

IF <A:DELIMITER>

88 VALUE <DELIM>

(null)

TRANSFORMATION:

OMITTED <- STANDARD

OCCURS <C:DELIM> <- OCCURS <B:DELIM> TO <C:DELIM>

<LITERAL> <- <LITERAL><DELIM>

FIGURE 6
SAMPLE TRANSFORMATION TABLE

NAME	PRIORITY	ACTION	TRANSFORM
TR0	90	00	<- ''
TR1	85	00	<LITERAL-LIST> <- '<DELIM>
TR2	75	00	<LITERAL-LIST> <- <LITERAL-LIST><DELIM>
TR3	80	00	<LITERAL> <- <LITERAL-LIST>'
TR4	70	15	DISPLAY <A:MESSAGE> UPON CRT <- DISPLAY <A:LITERAL> UPON TTY

(note: a non-zero ACTION denotes additional
procedural action)

5.1 Primitive Types

Specification of priority BNF begins with the definition of an entity:

5.1.0. An entity is a primitive which represents a sequence of one or more characters in the base language which is recognizable by the language's grammar as a legal string.

Identification of an entity is wholly dependent on the base language specifications. A sequence of entities may be grouped and treated as one. Such a grouping is called an entity collection:

5.1.1 An entity collection is a sequence of one or more entities which is treated as an a single entity.

5.2 Symbols

The relationship of the source text derived entities and the priority BNF elements is defined with the use of symbols:

5.2.0 A symbol is a primitive object; each symbol has both a value and a type. There exist three kinds of symbols in priority BNF: the terminal symbol, nonterminal symbol, and labeled nonterminal symbol.

5.2.1 A terminal symbol is a representation of a single entity. The terminal's value is the text representation of the specified entity; the terminal's type is undefined until active use.

5.2.2 A nonterminal symbol is a representation of an entity collection. The nonterminal's value is the text representation of the entity collection; the nonterminal's type is assigned upon creation by the transformation process and may change.

5.2.3 A labeled nonterminal symbol is a representation of an entity collection. The symbol can be represented by an ordered pair (a,b) where a is a label used to address the specific labeled nonterminal, and b is a nonterminal symbol.

5.3 Patterns

Combinations of the above symbols can be formed, types and values are initialized, and the result is a pattern:

5.3.0 A pattern is a sequence of zero or more symbols. Creation of a pattern implies the specification of initial types for nonterminal and labeled nonterminal symbols and the specification of values for terminal symbols.

Although the specification of a value for a terminal may seem to be in error considering that the terminal derives its value from the entity it represents, a pattern is used for matching and so must be initialized. This point is clarified with the explanation that the source text program entities are tokenized by the preprocessor and converted into tokens which are, in effect, terminal symbols. The symbols' values correspond to the entities' text representations. In addition, each token is assigned a default type of either delimiter, integer, or literal. Pattern matching is accomplished by comparing patterns to

the sequences of tokens. Both the tokens' types and values can be examined. The results of pattern matching can be the replacement of the tokens by any of the three primitive symbols previously defined. The definition used to perform this change is called a transformation:

5.3.1 A transformation (also known as a reduction) is an ordered pair (a,b) where a and b are both patterns. The b entity is called the matched (or replaced) pattern, and the a entry is the replacement pattern. A successful pattern match using b causes the replacement of the matched token string corresponding to b to be replaced by a . This replacement is denoted by " $a \leftarrow b$ ".

5.4 Sets

Now that patterns and transformations have been defined, the semantics involved in a reduction can be explicitly stated. To do so necessitates the definition of a few conventions to be used in the explanations:

5.4.0 Let XUY denote for sets X and Y , X union Y .

5.4.1 Let xy denote for sequences of symbols x and y , their concatenation.

5.4.2 The closure of a set A is denoted as A^* and is defined as

$$A^* = A[0]UA[1]UA[2]U...A[n]$$
 where each $A[i]$ is a set containing all possible combinations of i nonunique elements taken from the members of A .

5.4.3 Let T , N , and L denote sets of terminal, nonterminal, and labeled nonterminal symbols.

5.5 Transformation Classes

The following are classes of transforms allowable in priority ENF:

5.5.0. Any to Terminal: for all a and x such that $a \in T^*$, and $x \in (TUNUL)^*$, the transformation $a \leftarrow x$ causes the types and values of x to be replaced by the types and values of a .

5.5.1 Any to Nonterminal: for all A and x such that $A \in N$ and $x \in (TUNUL)^*$, the transformation $A \leftarrow x$ causes the types of x to be replaced by the type of A . The value of A is derived from the creation of an entity collection using the values of x . The type of A is as defined in the replacement pattern.

5.5.2. Labelnonterm to Labelnonterm: for all A, B, v, w, x , and y such that $v, w \in (TUNUL)^*$; $x, y \in (TUL)^*$; and $A, B \in L$ such that for $A = (p, q)$ and $B = (p, r)$, the transformation $xAy \leftarrow vBw$ is defined as follows:

1. the reductions $x \leftarrow v, y \leftarrow w, y \leftarrow v$, and $x \leftarrow w$ are as defined in 5.5.0, 5.5.1, and 5.5.2;
2. the reduction $A \leftarrow B$ causes the value of B to be assigned to value of A . The type of A is as defined in the replacement pattern.
3. the result of steps 1 and 2 cause the ultimate replacement of vBw by xAy .

5.5.3. Any to Mixed: for all A, w, x , and y such that $A \in N, w \in (TUNUL)^*$, and $x, y \in (TUL)^*$ the transformation $xAy \leftarrow w$ is defined as follows:

1. the reductions $x \leftarrow w$ and $y \leftarrow w$ are as defined in 5.5.0 5.5.2, and 5.5.3;
2. the reduction $A \leftarrow w$ causes the value of A to be conditionally assigned:
 - given the relative displacement $d[x]$ of A in xAy , the value of A is the entity collection of the values of the sequence of symbols $w[d[x]]$. . . $w[i]$ in w (i.e., beginning with the $d[x]$ th symbol in w). Symbol $w[i]$ is:
 - the final symbol in the sequence w ;
 - or
 - the symbol immediately preceding symbol $w[i+1]$

such that the value of $w[i+1]$ equals the value of the first symbol in sequence y (i.e., $y[1]$) and $y[1] \in T$.

if the second alternative can not be satisfied, the first alternative is used to define $w[i]$.

3. the result of steps 1 and 2 cause the replacement of w by xAy .

Note in particular rule 5.5.4; what the definition states is that a nonterminal symbol contained in a replacement pattern collects all the symbols in the matched pattern starting with the same relative displacement until either the string is exhausted or the nonterminal's following symbol's value matches a symbol value in the right pattern. This latter scheme works only when the following symbol is a terminal. Rule 5.5.1 is redundant when considered in the light of 5.5.3 but is introduced in the sake of clarity and continuity of the definitions.

5.6 Transformation Entries

A transformation is used by the user to form a transformation entry:

5.6.0. A transformation entry is a 5-tuple (a, b, c, d, e) where
 a is a character string denoting the entity's name,
 b is an integer denoting the entry's priority;
 c is an integer denoting an entry's procedural action upon match;
 d, e are patterns forming the transformation $d \leftarrow e$.

5.6.1. A transformation table is a set of transformation entries. The set may be null.

Priority values between entries are explicitly defined:

5.6.2. For transformation entries x and y ,
 $x = (a[x], b[x], c[x], d[x], e[x])$ and
 $y = (a[y], b[y], c[y], d[y], e[y])$,
 the priority of x is greater than the priority of y iff
 $|b[x]| > |b[y]|$.

A higher priority entry will always attempt a pattern match before a low priority entry.

5.7 Classification of Transforms and Priority Assignments

Experience with transformations has shown that their use in language conversion tends to present patterns of usage which can be categorized and analyzed. Transformations can be placed into three classes:

- simple transformation: a transformation which involves a single reduction which is achieved without the aid of any other transformation;
- group transformation: a set of transformations which work together to reduce or collect terminal symbols (i.e. tokens) into a single nonterminal symbol;
- compound transformation: a set of simple and group transformations which are coordinated to reduce a complex, variable length token string into a fixed

length, recognizable form which is finally reduced to the ultimate goal.

Use of isolated, simple transformations are dedicated to such tasks as elimination of optional strings and one step conversions of incompatible strings. The results are similar to that achieved by using a text editor upon source text. The simple isolated transform is usually (though not necessarily) of the form terminal-to-terminal (5.5.1).

Examination of the group transformation displays a decomposition into three parts:

- initialization reduction: produces a result which keys the building of the nonterminal by the rest of the group;
- collection reduction: collects a string of tokens into a nonterminal symbol one token at a time;
- termination reduction: terminates the nonterminal collection by producing as a result a token string which can not be matched by any transformation in the group.

The group transformation is the tool which gives power to the macro language. The difficulty present with most macro languages (dealing with variable length macro call parameter lists by both text value and syntactic type) has been resolved by the use of the group transform. The

reduction of such an arbitrary length string to a single symbol allows simple but powerful text manipulation and generation.

Each of the three reduction parts of the group transform is itself a simple transform. The initialization reduction of a group transformation is usually of the form any-to-nonterminal (5.5.1) as is the collection reduction. The termination reduction is usually of the form labelnonterm-to-labelnonterm (5.5.2).

Compound transformations have initialization, collection, and termination reductions as well. The collection reduction for a compound transform is a set of zero or more group transforms; the initialization and termination reductions are typically zero or more simple transformations, although compound transforms may be used. The definition of a collection reduction as previously given must be expanded when applied to the compound transform to allow the production of a resulting set of nonterminal symbols. This is because a typical compound transformation operates upon a complex, variable length statement which must have its subparts each reduced to a single nonterminal symbol before the termination reduction can recognize the statement and produce the ultimate result. The initialization reduction is optional; when present, it is usually applied to the elimination of optional words so as to "standardize" the statement for manipulation by other reductions. The termination reduction, however, must always

be present.

Examples of the transformation classes can be viewed in Figure 6. All five entries can be termed simple transformations as each entry performs a text transformation unaided. A group transform exists in the table due to entries TR1, TR2, and TR3. TR1 is the initialization reduction, TR2 is the collection reduction, and TR3 is the termination reduction. Together the entries reduce an arbitrary sequence of symbols bounded by quotation marks into a single nonterminal. This group transform in turn is the collection reduction of the compound transformation TR0 through TR4. Note that TR0 serves as the initialization reduction, and TR4 is the termination reduction. The relative priorities between the table entries will be elaborated upon in the following paragraphs.

Assignment of priorities to individual transformations is directly dependent upon their roles in the larger scheme of the simple, group, and compound classes previously described. Use of the macro language implies a correct use, that is, the macro writer intends that his transformation table produces the results which he desires. These results are dependent upon the syntax, semantics, and priority assignment in the priority BNF macro language. Syntax is defined in the users manual; the semantics of the individual transformation has been previously defined; and the semantics of priority numbers of any given pair of transformations has been also defined (5.6.2). The

remaining task is the intelligent assignment of priorities to transformations to produce predictable (and thus, correct) results. Although such an issue is as nebulous to define as an attempt to define how to construct a correct program using FORTRAN, a trio of rules are presented for the classes of transformations previously listed. Use of these rules when constructing an instance of a transformation class guarantees a predictable result which can be used by the programmer to produce a correct result:

1. In a group transformation, the termination reduction must have a higher priority than the collection reduction;
2. In a compound transformation, the initialization reduction must have a higher priority than the termination reduction;
3. All other priority assignments can be determined only within the context in which the transforms are applied.

It must be noted that whenever an initialization or termination reduction consists of more than a single transformation entry, the priority of the set of entries is taken to be the minimum priority of the entries within that set. A null initialization reduction is considered to have maximum priority.

Rule one states that a group transformation must terminate. This can only be accomplished by locating the termination string before continuing the collection. Rule

two states that no compound transformation can execute until the string to be operated upon is placed into a recognizable form. Rule three implies that simple, isolated transformations may have arbitrary values; the topic of interactions between transformation classes is beyond the scope of this report and is the responsibility of the programmer.

Examples of application of these rules are given in section VI of the users manual [19]. The users manual also expands upon the topic of transformation classes by defining their relationship to device efficiency and action routine application.

6.0 Evaluation and Conclusions

Chapter two of this report was concerned with the various features and facilities of a macroprocessor as applied to general purpose use. In addition, a list of desired objectives was presented for the soon-to-be implemented device. Construction and use of the preprocessor now allows a critical evaluation of the device versus the standards previously mentioned.

If one general statement about the device's capabilities can be made, it must be this: the macro calls as defined by priority BNF are a simple but powerful set of tools for text translation. The mechanics of these tools are straightforward, clear, and precisely defined. No ambiguity or hidden details exist in their use. Such a set of calls, in turn, gives the user great potential power to apply the device in a wide range of situations with credible results.

Evaluation of the macro calls and their associated action routines versus the standards set by McIlroy [20] and Brown [21] is stated following. A strong application of the desired feature is followed by "(+)", as acceptable application is denoted by "{0}", and a weak application is shown by "(-)". McIlroy's list is presented first.

-- use of nested calls: the preprocessor scans source text one sentence at a time, evaluating and

reevaluating until no possible macro call can be applied (+);

-- use of conditional calls: the macro expansion is fixed in priority BNF; an action routine must be written to override the transformation mechanism and resubstitute the original text (-);

-- creation of source text symbols: can be done in the macro call itself or can be accomplished via an action routine which must access tokenized keywords input with the macro deck at program's beginning (0);

-- grouping (precedence) of macro call parameters: accomplished explicitly through the use of the group transformation (+);

-- nested macro definitions: not allowed (-);

-- recursive calls: accomplished differently than as normally described, but is easily used (+).

Brown's list is more correctly a listing of macroprocessor features rather than an evaluation. Prejudicial ratings are given to denote the quality of the features implemented.

-- calling syntax: notation independent, syntax driven; allows for a wide variety of implementable macro calls (+);

-- text evaluation: parameters maintain a delayed evaluation, i.e., a call by name (+); there exists an

extensive use of macro-time variables and tables by action routines (+); text symbols may be created but no check is made for their uniqueness (-);

-- macro-time statements: refer to the action routines of this implementation; Action routines are coded in the language of the device (PASCAL) and their statements show a generality and power (+). Action routines must be inserted internally into the preprocessor body (-);

-- implementation: internal storage is maintained via contiguous lists and linked lists and allows a high degree of text manipulation (0); the pattern matching sequence is slow and involves little optimization (-); dynamic allocation of storage is allowed (+);

-- user considerations:

1. macro definition: macro calls are somewhat difficult to use for a user not exposed to formal language notation (0); action routine coding is simple, and the macro-time language is easy to use; action routine coding is supplemented by built-in text construction subroutines (+);
2. error detection: minimal syntax error detection and no recovery (-); internal device error detection is limited to table and file overflow, although extensive tracing features are provided for debugging transformation entries.

The PRECCB preprocessor seems strongest in its definition and use of the macro call and weakest in execution speeds and internal response to abnormal conditions. The former point is (hopefully) an indication of good planning and solid theory (and much of this was laid in James Bell's work with Proteus); the latter is a manifestation of the pressures a tight project schedule produces. No feature of the preprocessor seems so serious as to fault the device as a whole, in fact, the groundwork seems to have been laid for the production of a wide range of specific macroprocessors based on this general purpose design.

In addition to the lists of McIlroy and Brown, a self-created list was also introduced in chapter two as a checklist to be used when comparing the implemented device versus specific motivational need: the conversion of IBM CCBOL. The needs of that list seem to have been met in all accounts. The constructed device is more than adequate for the high level language conversion task. Initial use has proved this point to be so; no major problems have been encountered with the device.

6.1 Extensibility and Future Uses

As previously described, one of the objectives in

constructing the PRECOB preprocessor was that its use would not be limited to CCBOL to COBOL conversion alone. The device was to be designed so to provide a framework for a wide variety of applications; the ultimate goal was to produce a general purpose macroprocessor. Although the results seem to have fallen short in some areas (witness the previous evaluations), the facilities that are provided allow application to wide areas of use. Some of these are:

-- source language conversion: since the actual conversion specifications are contained within the macro definition table, conversion of any host to target language involves only the changing of the macro table. Formatting procedures within the device may have to be altered as well, but this involves only the removal of a small number of subroutines and their replacement by modules compatible with the chosen host and target languages;

-- creation of an extensible language: the device could be incorporated into the front end of an existing compiler to allow the creation of an extensible language.

-- compiler generation: since the preprocessor contains facilities for lexical and syntactic analysis, the device could be treated as a "compiler-compiler". The user would specify legal language constructs via priority BNF, insert his grammar into the macro table,

and write action routines which would cause code generation or interpretation.

-- text generation: items that are conditionally generated and inserted into existing source code (e.g., debug procedures, repetitive code production, or library routines) could be handled by the preprocessor. This function is best exemplified by the preprocessor present in PL/I.

-- pattern recognition: the macro table could be set up to scan for specific occurrences of designated character strings. Since the device is syntax driven, a higher degree of complexity could be incorporated than found in a text editor: strings could be assigned types, and recognition of type patterns could be achieved.

-- text editing: as mentioned before, the preprocessor has powerful text editing capabilities.

Certainly more applications exist; these are but a few. The device's capability for easy modification of existing internal code make the possibilities for extensibility wide. As an example, the tokenizing routines in the current implementation have been removed to make way for a new version that supports input containing as many as 4500 unique character strings. The modification only took forty man-hours of time (although they were the designer's). As changes are incorporated into the processor's own base

language, PASCAL, the device can grow along with its language. In fact, the most severe restriction encountered at this point has been the implementation of PASCAL in operation. Such difficulties are transient and will not effect the preprocessor's successful application to a wide range of areas.

REFERENCES

1. Brown, P. J., "A Survey of Macroprocessors", Annual Review of Automatic Programming, Pergammon Press, London, 1969, page 38.
2. Cheatham, T. E., "The Introduction of Definitional Facilities into Higher Level Languages", Proceedings of the AFIPS, 1966, pp. 623-637.
3. Leavenworth, R. M., "Syntax Macros and Extended Translation", Communications of the ACM 6 (November 1966): 709-93.
4. McIlroy, M. D., "Macro Instruction Extension of Compiler Languages", Communications of the ACM 3 (April 1960): 219.
5. Brown, pp. 48-83.
6. Brown, page 63.
7. IBM Corporation, PL/I(F) Language Reference Manual, publication number GC28-8201-4, White Plains, New York, 1970, pp. 205-213.
8. Griswold, R. E., Poage, J. F., and Polonsky, I. P., The SNOBOL4 Programming Language, Second Edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.
9. McIlroy, pp. 219-218.
10. Brown, P. J., "The ML/I Macroprocessor", Communications of the ACM 10 (October 1967): 618-623.
11. Bell, J. R., "The Design of a Minimal Expandable Computer Language", doctoral dissertation, Computer Science Department, Stanford University, Stanford, California, 1968.
12. Bell, pp. 168-207.
13. Brinch Hansen, P., "Sequential PASCAL Report", Information Science, California Institute of Technology, Pasadena, California, 1972.
14. Hartmann, Alfred, "A Concurrent PASCAL Compiler for Minicomputers", doctoral dissertation, Information Science, California Institute of Technology, Pasadena, California, 1976, pp.17-23.

15. Schmidt, David, "User Guide to the PRECOB Text Preprocessor", technical report number CS77-16, Computer Science Department, Kansas State University, Manhattan, Kansas, 1977.
16. Brinch Hansen, E., "Concurrent PASCAL Implementation Notes", Information Science, California Institute of Technology, Pasadena, California, 1975.
17. Schmidt, pp. 24-31.
18. Bell, pp. 1-167.
19. Schmidt, pp. 167-172.
20. McIlroy, pp. 215-218.
21. Brown, "A Survey of Macroprocessors", pp. 40-76.

BIBLIOGRAPHY

- Fell, James R., "The Design of a Minimal Expandable Computer Language". Doctoral dissertation, Computer Science Department, Stanford University, Stanford, California, 1968.
- Bennett, Richard K., and Neumann, David H. "Extension of Existing Compilers by Sophisticated Use of Macros". Communications of the ACM 7 (September 1964): 541-42.
- Erinch Hansen, P. "Concurrent PASCAL Implementation Notes", Information Science, California Institute of Technology, Pasadena, California, 1975.
- . "Sequential PASCAL Report". Information Science, California Institute of Technology, Pasadena, California, 1972.
- Brown, P. J. "A Survey of Macroprocessors". Annual Review of Automatic Programming, Pergamon Press, London, 1969, 6,2.
- . Macroprocessors and Techniques for Portable Software. John Wiley and Sons, New York, 1974.
- . "The ML/I Macroprocessor". Communications of the ACM 10 (October 1967): 618-623.
- Campbell-Kelly, M. An Introduction to Macros. American Elsevier, New York, 1973.
- Cheatham, T. E. "The Introduction of Definitional Facilities into Higher Level Languages". Proceedings of the AFIPS, 1966, FJCC 29: 623-637.
- Cole, A.J. Macroprocessors. Cambridge University Press, New York, 1976.
- Conway, M. E. "Design of a Separable Transition-Diagram Compiler". Communications of the ACM 6 (July 1963): 283-306.
- Early, J. "Towards an Understanding of Data Structures". Communications of the ACM 14 (October 1971): 617-27.
- Garwick, J. V. "GPL, a Truly General Purpose Language". Communications of the ACM 9 (September 1968): 634-638.
- Gries, David. Compiler Construction for Digital Computers. John Wiley and Sons, New York, 1971, pp. 11-48, 154-169.

- Griswold, R. E., Peage, J. F., and Polonsky, I. P. The SNCBOL4 Programming Language, Second Edition. Prentice-Hall, Englewood Cliffs, New Jersey, 1971.
- Halpern, M. I. "Toward a General Processor for Programming Languages". Communications of the ACM 11 (January 1968): 15-25.
- Hartmann, Alfred E. "A Concurrent PASCAL Compiler for Minicomputers". Doctoral dissertation, Information Science, California Institute of Technology, Pasadena, California, 1976.
- Interdata, incorporated. OS32-MT Program Reference Manual. Publication number 29-390R04, Cceanport, New Jersey, 1976.
- IBM Corporation. PL/I(F) Language Reference Manual. Publication number GC28-8201-4, White Plains, New York, 1970.
- Irons, E. T. "Experience with an Extensible Language". Communications of the ACM 13 (January 1970): 131-140.
- Knuth, Donald E. The Art of Computer Programming, Volume One. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.
- Leavenworth, R. M. "Syntax Macros and Extended Translation". Communications of the ACM 9 (November 1966): 790-793.
- McIlroy, M. D. "Macro Extension of Compiler Languages". Communications of the ACM 3 (April 1960): 214-220.
- Schmidt, David. "User Guide to the PRECOB Text Preprocessor". Technical report number CS77-16, Computer Science Department, Kansas State University, Manhattan, Kansas, 1977.
- Solntseff, N., and Yezeriski, A. "A Survey of Extensible Programming Languages". Annual Review in Automatic Programming, Pergamon Press, New York, 1974, 7,2.
- Strachey, C. "A General Purpose Macroprocessor". Computer Journal 8 (October 1965): 225-241.
- Waite, W. M. "A Language Independent Macroprocessor". Communications of the ACM 10 (July 1967): 433-441.
- Wilkes, M. V. "An Experiment with a Self Compiling Compiler". Annual Review in Automatic Programming, Pergamon Press, Oxford, 1964, 4: 1-48.

APPENDIX A
BREAKDOWN OF CONSTRUCTION EFFORT BY PHASES

PHASE	MAN-HOURS
Phase 1: Conversion of basic device to PASCAL code	180*
Phase 2: Implementation of formatting routines	132
Phase 3: Conversion to two passes	93
Phase 4: Coding of COBOL transformations and action routines	47
Phase 5: Composition of external documentation and users guide	91
Total	543 man-hours

* includes eighty hours of effort from Gary Anderson, a graduate student to whom I am indebted for valuable assistance and advice.

APPENDIX B
PREPROCESSOR INPUT FOR
INTERDATA COBOL CONVERSION

```

00001
00002
00003 PATTEND DELIM ANY <"@: ^ ^0 TAB SPACE LITERAL
00004 LEVEL-NO ID %7 %8 %12 %20 KEYWORD INT COND
00005 OR NOT AND ACCEPT ADD ALTER CALL CLOSE COMPUTE DISPLAY
    DIVIDE EXAMINE
00006 EXHIBIT GENERATE GO GOBACK IF INITIATE MOVE MULTIPLY O
PEN PERFORM READ
00007 READY RELEASE RESET RETURN REWRITE SEARCH SET SORT STA
RT STOP SUBTRACT
00008 TRANSFORM WRITE INDEXED PIC PICTURE VALUE ONE
00009 SPACES INTO FROM GIVING BY TO UP THEN ELSE UNTIL THRU
DOWN ZERO LEVEL-88
00010 FINISHED NEXT ^1
00011 ALL OCCURS TIMES
00012 '
00013 TR00 00091 00000
00014 "THRU" ^ "THROUGH"
00015 TR1 00092 00000
00016 "^1" ^ ", "
00017 TR2 00092 00000
00018 ^ "; "
00019 TR3 00091 00000
00020 "" ^ "EJECT"
00021 TR7 00069 00000
00022 "RECORDING" ^ "RECORDING MODE"
00023 TR8 00068 00000
00024 "" ^ "RECORDING <ANY>"
00025 TR9 00091 00000
00026 ^ "IS"
00027 TR10 00091 00000
00028 ^ "ARE"
00029 TR101 00070 00007
00030 ^ "DEPENDING <ANY>"
00031 TR102 00091 00000
00032 ^ "AT"
00033 TR103 00091 00000
00034 ^ "SKIP1"
00035 TR104 00091 00000
00036 ^ "SKIP3"
00037 TR105 00065 00099
00038 ^ "ASCENDING KEY <DELIM>"
00039 TR106 00065 00099
00040 ^ "DESCENDING KEY <DELIM>"
00041 TR11 00070 00000
00042 "OMITTED" ^ "STANDARD"
00043 TR14 00070 00000
00044 "SET <-A:ANY> TO ^1 <-B:ANY>" ^ "SET <-A:ANY> TO + <-B:A
NY>"
00045 TR145 00071 00000
00046 "DEPENDING" ^ "DEPENDING ON"
00047 TR16 00070 00099
00048 "OCCURS <-C:ANY>" ^ "OCCURS <-B:ANY> TO <-C:ANY>"

```



```

00049 TR17 00070 00000
00050 "INTERDATA MODEL-2-32" ^ "ISM-370"
00051 TR21 00091 00000
00052 "ZERO" ^ "ZEROS"
00053 TR22 00091 00000
00054 ^ "ZEROS"
00055 TR24 00091 00000
00056 "VALUE" ^ "VALUES"
00057 TR2405 00091 00000
00058 "VALUE ^1 <-A:ANY>" ^ "VALUE + <-A:ANY>"
00059 TR2406 00090 00000
00060 "THRU ^1 <-A:ANY>" ^ "THRU + <-A:ANY>"
00061 TR25 00090 00088
00062 "88" ^ "88 <ID>"
00063 TR26 00089 00099
00064 "" ^ "88 VALUE J"
00065 TR27 00088 00087
00066 "88 VALUE" ^ "88 VALUE <ANY> THRU <ANY>"
00067 TR28 00087 00086
00068 ^ "88 VALUE <ANY>"
00069 TR29 00070 00065
00070 "<-A:COND>" ^ "<-A:LEVEL-88>"
00071 TR295 00005 00005
00072 "INDEXED BY <-A:INDEX>" ^ "INDEXED BY <-A:DELIM>"
00073 TR294 00010 00007
00074 "<OCCURS>" ^ "OCCURS <-A:ANY> TIMES"
00075 TR295 00022 00015
00076 "SEARCH <-A:SEARCH>" ^ "SEARCH ALL <-A:ID>"
00077 TR31 00022 00000
00078 "SEARCH <-A:SEARCH>" ^ "SEARCH <-A:ID>"
00079 TR32 00019 00010
00080 ^ "SEARCH<-A:SEARCH> VARYING <-B:ANY>"
00081 TR34 00019 00000
00082 "SEARCH <-A:SEARCH><-B:ENDLIST>" ^ "SEARCH<-A:SEARCH>END
<-B:ANY>"
00083 TR35 00020 00000
00084 " <ENDLIST>" ^ "<ENDLIST> <ANY>"
00085 TR36 00021 00011
00086 "WHEN" ^ "<ENDLIST> WHEN"
00087 TR37 00018 00000
00088 "<-A:CONDLIST>" ^ "WHEN <-A:ANY>"
00089 TR38 00017 00000
00090 "<CONDLIST>" ^ "<CONDLIST><ANY>"
00091 TR39 00019 00013
00092 "<-B:WHENLIST>" ^ "<-A:CONDLIST><-B:KEYWORD>"
00093 TR41 00016 00012
00094 ^ "<-A:WHENLIST> <-B:WHENLIST>"
00095 TR40 00015 00000
00096 "<WHENLIST>" ^ "<WHENLIST> <ANY>"
00097 TR42 00016 00012
00098 "J" ^ "<-A:WHENLIST>J"
00099 TR43 00018 00014

```

```

00100  "MOVE 0 TO FINISHED(0+0) %12 X PERFORM SEARCH+0+0 UNTI
L FINISHED(0+0) = 1]"
00101  ^ "SEARCH <SEARCH>]"
00102  TR44  00080 00005
00103  "01 FTABLE X %12 FINISHED PIC 9 OCCURS 99 TIMES X %2 X
" ^
00104  "PROCEDURE DIVISION"
00105  TR45  00080 00005
00106  ^ "LINKAGE SECTION"
00107  TR47  00091 00000
00108  "<OP>"^"="
00109  TR4701 00092 00000
00110  ^ "NOT ="
00111  TR48  00091 00000
00112  "<LOP>"^"AND"
00113  TR49  00091 00000
00114  ^"OR"
00115  TR50  00079 00000
00116  "<-A:ANY><-B:OP><-C:ANY><-D:LOP> %20 <-A:ANY><-B:OP><-
E:ANY><-F:LOP>" ^
00117  "<-A:ANY><-B:OP><-C:ANY><-D:LOP><-E:ANY><-F:LOP>"
00118  TR51  00079 00000
00119  "<-A:ANY><-B:OP><-C:ANY><-D:LOP> %20 <-A:ANY><-B:OP><-
E:ANY> %20 <-F:KEYWORD>"
00120  ^
00121  "<-A:ANY><-B:OP><-C:ANY><-D:LOP><-E:ANY><-F:KEYWO
RD>"
00122  TR52  00060 00000
00123  "PERFORM <PARALIST>"^"PERFORM<DELIM> THRU <DELIM>"
00124  TR53  00055 00000
00125  ^"PERFORM<DELIM>"
00126  TR54  00047 00020
00127  "<-A:UNTIL><-A:PCONDLIST>"^"UNTIL<-A:ANY>"
00128  TR55  00050 00000
00129  "<PCONDLIST>"^"<PCONDLIST><ANY>"
00130  TR56  00052 00000
00131  "<-A:PCOND><-B:KEYWORD>"^"<-A:PCONDLIST><-B:KEYWORD>"
00132  TR57  00045 00021
00133  "MOVE<-C:DDD>TO<-D:ID>%12 PERFORM PERFORM+0+0 <-B:UNTI
L> FINISHED(0+0) = 1"
00134  ^
00135  "PERFORM<-A:PARALIST>VARYING<-D:ID>FROM<-C:ANY>BY<ANY>
<-B:UNTIL><PCOND>"
00136  TR58  00044 00022
00137  "MOVE 0 TO FINISHED(0+0) %12 MOVE <-A:DDD> TO <-B:ID>"

00138  ^
00139  "MOVE <-A:DDD> TO <-B:ID>"

```

NAME PRIORITY ACTION TRANSFORM

```

TR00  00091 00000 THRU <-- THROUGH
TR1   00092 00000 ^1 <-- ,
TR2   00092 00000 ^1 <-- !
TR3   00091 00000 <-- EJECT
TR7   00069 00000 RECORDING <-- RECORDING MODE
TR8   00068 00000 <-- RECORDING <ANY>
TR9   00091 00000 <-- IS
TR10  00091 00000 <-- ARE
TR101 00070 00007 <-- DEPENDING <ANY>
TR102 00091 00000 <-- AT
TR103 00091 00000 <-- SKIP1
TR104 00091 00000 <-- SKIP3
TR105 00065 00099 <-- ASCENDING KEY <DELIM>
TR106 00065 00099 <-- DESCENDING KEY <DELIM>
TR11  00070 00000 OMITTED <-- STANDARD
TR14  00070 00000 SET <-A:ANY> TO ^1 <-B:ANY> <-- SET <-A:A
NY> TO + <-B:ANY>
TR145 00071 00000 DEPENDING <-- DEPENDING ON
TR16  00070 00099 OCCURS <-C:ANY> <-- OCCURS <-B:ANY> TO <-
C:ANY>
TR17  00070 00000 INTERDATA MODEL-8-32 <-- IBM-370
TR21  00091 00000 ZERO <-- ZERGES
TR22  00091 00000 ZERO <-- ZEROS
TR24  00091 00000 VALUE <-- VALUES
TR2405 00091 00000 VALUE ^1 <-A:ANY> <-- VALUE + <-A:ANY>
TR2406 00090 00000 THRU ^1 <-A:ANY> <-- THRU + <-A:ANY>
TR25  00090 00088 88 <-- 88 <ID>
TR26  00089 00099 <-- 88 VALUE ]
TR27  00088 00087 88 VALUE <-- 88 VALUE <ANY> THRU <ANY>
TR28  00087 00086 88 VALUE <-- 88 VALUE <ANY>
TR29  00070 00085 <-A:COND> <-- <-A:LEVEL-88>
TR293 00005 00006 INDEXED BY <-A:INDEX> <-- INDEXED BY <-A:
DELIM>
TR294 00010 00007 <OCCURS> <-- OCCURS <-A:ANY> TIMES
TR295 00022 00015 SEARCH <-A:SEARCH> <-- SEARCH ALL <-A:ID>

TR31  00022 00009 SEARCH <-A:SEARCH> <-- SEARCH <-A:ID>
TR32  00019 00010 SEARCH <-A:SEARCH> <-- SEARCH <-A:SEARCH>
VARYING <-B:ANY>
TR34  00019 00000 SEARCH <-A:SEARCH> <-B:ENDLIST> <-- SEARC
H <-A:SEARCH> END <-B:ANY>
TR35  00020 00000 <ENDLIST> <-- <ENDLIST> <ANY>
TR36  00021 00011 WHEN <-- <ENDLIST> WHEN
TR37  00018 00000 <-A:CONDLIST> <-- WHEN! <-A:ANY>
TR38  00017 00000 <CONDLIST> <-- <CONDLIST> <ANY>
TR39  00019 00013 <-B:WHENLIST> <-- <-A:CONDLIST> <-B:KEY&O
RD>
TR41  00016 00012 <-B:WHENLIST> <-- <-A:WHENLIST> <-B:WHENL
IST>

```

```

TR40 00015 00000 <WHENLIST> <-- <WHENLIST> <ANY>
TR42 00016 00012 J <-- <-A:WHENLIST> J
TR43 00018 00014 MOVE 0 TO FINISHED ( 0 + 0 ) %12 X PERFORM
SEARCH + 0 + 0 UNTIL FINISHED ( 0 + 0 ) = 1 J <-- SEARCH <S
EARCH> J
TR44 00080 00005 01 FTABLE X %12 FINISHED PIC 9 OCCURS 99 I
IRLS X %8 X <-- PROCEDURE DIVISION
TR45 00080 00005 01 FTABLE X %12 FINISHED PIC 9 OCCURS 99 I
IRLS X %8 X <-- LINKAGE SECTION
TR47 00091 00000 <OP> <-- =
TR4701 00092 00000 <OP> <-- NOT =
TR48 00091 00000 <LOP> <-- AND
TR49 00091 00000 <LOP> <-- OR
TR50 00079 00000 <-A:ANY> <-B:OP> <-C:ANY> <-D:LOP> %20 <-A
:ANY> <-B:OP> <-E:ANY> <-F:LOP> <-- <-A:ANY> <-B:OP> <-C:ANY
> <-D:LOP> <-E:ANY> <-F:LOP>
TR51 00079 00000 <-A:ANY> <-B:OP> <-C:ANY> <-D:LOP> %20 <-A
:ANY> <-B:OP> <-E:ANY> %20 <-F:KEYWORD> <-- <-A:ANY> <-D:LOP>
<-C:ANY> <-D:LOP> <-E:ANY> <-F:KEYWORD>
TR52 00060 00000 PERFORM <PARALIST> <-- PERFORM <DELIM> IN
RU <DELIM>
TR53 00055 00000 PERFORM <PARALIST> <-- PERFORM <DELIM>
TR54 00047 00020 <-A:UNTIL> <-A:PCONDLIST> <-- UNTIL <-A:A
NY>
TR55 00050 00000 <PCONDLIST> <-- <PCONDLIST> <ANY>
TR56 00052 00000 <-A:PCOND> <-B:KEYWORD> <-- <-A:PCONDLIST>
> <-B:KEYWORD>
TR57 00045 00021 MOVE <-C:DOD> TO <-D:ID> %12 PERFORM PERFO
RM + 0 + 0 <-E:UNTIL> FINISHED ( 0 + 0 ) = 1 <-- PERFORM <-A
:PARALIST> VARYING <-D:ID> FROM <-C:ANY> BY <ANY> <-B:UNTIL>
<PCOND>
TR58 00044 00022 MOVE 0 TO FINISHED ( 0 + 0 ) %12 MOVE <-A:
DOD> TO <-B:ID> <-- MOVE <-A:DOD> TO <-B:ID>

```

```

00001      EJECT
00002      IDENTIFICATION DIVISION.
00003      PROGRAM-ID. TEST.
00004      ENVIRONMENT DIVISION.
00005      CONFIGURATION SECTION.
00006      SOURCE-COMPUTER. IBM-370.
00007      OBJECT-COMPUTER. IBM-370.
00008      INPUT-OUTPUT SECTION.
00009      FILE-CONTROL.
00010          SELECT AFIL ASSIGN TO SYS003-UT-2314-S.
00011      EJECT
00012      DATA DIVISION.
00013      FILE SECTION.
00014      FD AFIL
00015          LABEL RECORDS ARE STANDARD
00016          RECORDING MODE IS FIXED
00017          DATA RECORD IS AREC.
00018      01 AREC PIC X(80).
00019      WORKING-STORAGE SECTION.
00020      77 J PIC 9.
00021      77 STUDENT PIC 9.
00022          88 FROSH VALUE IS 1.
00023          88 SOPH VALUE 2.
00024          88 JR VALUE 3, 6.
00025          88 UPPER-GRAD VALUES ARE 4 THRU 7.
00026      77 X PIC 9.
00027      77 SEX PIC 99.
00028          88 MALE VALUES 5 THRU 18.
00029          88 FEMALES VALUES 19 THRU 35.
00030      77 FFF PIC X(6).
00031          88 STARS VALUE ALL '*'.
00032      01 TABLE-SECTION.
00033          02 TABLE OCCURS 3 TO 11 TIMES DEPENDING ON
00034              J
00035              INDEXED BY AA.
00036              03 ITEM1 PIC XX.
00037          02 TABLEC OCCURS 4 TIMES INDEXED BY B.
00038              03 NAME PIC X(4).
00039          02 TABLED OCCURS 2 TIMES INDEXED BY C.
00040              03 DITEM PIC X(5).
00041          02 TABLEA OCCURS 10 TIMES INDEXED BY A.
00042              03 B PIC 9.
00043      PROCEDURE DIVISION.
00044          IF STARS THEN DISPLAY '*****'.
00045          IF FROSH THEN MOVE 1 TO X.
00046          IF JR THEN MOVE '2' TO X.
00047          DISPLAY '2'.
00048          SET B TO +2.
00049          SEARCH TABLEA VARYING C
00050              AT END MOVE 1 TO X
00051              WHEN B(A) = 0 GO TO PARA-A
00052              WHEN B(A) = 1 GO TO PARA-B.
00053          SEARCH ALL TABLEC WHEN ITEM1 (AA) > 2 STOP

```

```
      RUN.
00053      SEARCH ALL TABLEA WHEN B (A) = 4 STOP RUN.
00054      IF A = 1 OR 2 OR 3 THEN STOP RUN.
00055      IF MALES THEN STOP RUN.
00056      SET I TO 77.
00057      SEARCH TABLED
00058      AT END NEXT SENTENCE
00059      WHEN I = 2 STOP RUN
00060      WHEN I = 4 GO TO PARA-A.
00061      PERFORM PARA-A THRU PARA-B VARYING X FROM 4
      BY 2
00062      UNTIL X EQUALS 9.
00063      PARA-A. EXIT.
00064      PARA-B. EXIT.
00065      PERFORM PARA-A UNTIL J = 0.
00066      STOP RUN.
00067
```

APPENDIX C
SAMPLE PREPROCESSOR OUTPUT
(using previous input)

```

000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. TEST.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. INTERDATA MODEL-6-32.
000060 OBJECT-COMPUTER. INTERDATA MODEL-6-32.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT AFIL ASSIGN TO SYS003-UT-2314-S.
000100 DATA DIVISION.
000110 FILE SECTION.
000120 FD AFIL
000130     LABEL RECORDS OMITTED
000140     DATA RECORD AREC.
000150 01 AREC PIC X(80).
000160 WORKING-STORAGE SECTION.
000170 77 J PIC 9.
000180 77     STUDENT PIC 9.
000190*     88 FROSH VALUE IS 1.
000200*     88 SOPH VALUE 2.
000210*     88 JR VALUE 3, 6.
000220*     88 UPPER-GRAD VALUES ARE 4 THRU 7.
000230 77 X PIC 9.
000240 77 SEX PIC 99.
000250*     88 MALE VALUES 5 THRU 18.
000260*     88 FEMALES VALUES 19 THRU 35.
000270 77 FFF PIC X(6).
000280*     88 STARS VALUE ALL '*'.
000290 01 TABLE-SECTION.
000300*     02 TABLEB OCCURS 3 TO 11 TIMES DEPENDING ON J
000310*     INDEXED BY AA.
000320     02 TABLEC OCCURS 11 TIMES
000330     INDEXED BY AA.
000340     03 ITEM1 PIC XX.
000350     02 TABLED OCCURS 4 TIMES INDEXED BY B.
000360     03 NAME PIC X(4).
000370     02 TABLEE OCCURS 2 TIMES INDEXED BY C.
000380     03 DITEM PIC X(5).
000390     02 TABLEF OCCURS 10 TIMES INDEXED BY A.
000400     03 B PIC 9.
000410 01 FTABLE.
000420     FINISHED PIC 9 OCCURS 99 TIMES.
000430 PROCEDURE DIVISION.
000440*     IF STARS THEN DISPLAY '*****'.
000450*     IF FFF = ALL '*'
000460*         THEN DISPLAY '*****'.
000470*     IF FROSH THEN MOVE 1 TO X.
000480*     IF STUDENT = 1
000490*         THEN MOVE 1 TO X.
000500*     IF JR THEN MOVE '2' TO X.
000510*     IF STUDENT = 3
000520*         OR STUDENT = 6

```



```

000530          THEN MOVE '2' TO X.
000540      DISPLAY '2'.
000550      SET B TO 2.
000560*      SEARCH TABLEA VARYING C
000570*          AT END MOVE 1 TO X
000580*          WHEN B(A) = 0 GO TO PARA-A
000590*          WHEN B(A) = 1 GO TO PARA-B.
000600      MOVE 0 TO FINISHED ( 01 )
000610      SET C TO 1
000620      PERFORM SEARCH01 UNTIL FINISHED ( 01 ) = 1.
000630*      SEARCH ALL TABLES WHEN ITEM1 (AA) > 2 STOP RUN.
000640      MOVE 0 TO FINISHED ( 02 )
000650      SET AA TO 1
000660      PERFORM SEARCH02 UNTIL FINISHED ( 02 ) = 1.
000670*      SEARCH ALL TABLEA WHEN B (A) = 4 STOP RUN.
000680      MOVE 0 TO FINISHED ( 03 )
000690      SET A TO 1
000700      PERFORM SEARCH03 UNTIL FINISHED ( 03 ) = 1.
000710          IF A = 1 OR
000720              A = 2 OR
000730              A = 3
000740              THEN STOP RUN.
000750      IF MALES THEN STOP RUN.
000760      SET I TO 77.
000770*      SEARCH TABLED
000780*          AT END NEXT SENTENCE
000790*          WHEN I = 2 STOP RUN
000800*          WHEN I = 4 GO TO PARA-A.
000810      MOVE 0 TO FINISHED ( 04 )
000820      PERFORM SEARCH04 UNTIL FINISHED ( 04 ) = 1.
000830*      PERFORM PARA-A THRU PARA-B VARYING X FROM 4 BY 2
000840*          UNTIL X EQUALS 9.
000850      MOVE 0 TO FINISHED ( 05 )
000860      MOVE 4 TO X
000870      PERFORM PERFORM05 UNTIL FINISHED ( 05 ) = 1.
000880      PARA-A. EXIT.
000890      PARA-B. EXIT.
000900      PERFORM PARA-A UNTIL J = 0.
000910      STOP RUN.
000920      SEARCH01.
000930          IF A > 10
000940              MOVE 1 TO FINISHED ( 01 )
000950              MOVE 1 TO X
000960          ELSE IF B(A) = 0
000970              MOVE 1 TO FINISHED ( 01 )
000980              GO TO PARA-A
000990          ELSE IF B(A) = 1
001000              MOVE 1 TO FINISHED ( 01 )
001010              GO TO PARA-B
001020          ELSE SET A UP BY 1
001030              SET C UP BY 1.
001040      SEARCH02.
001050          IF AA > J

```

```

001060             MOVE 1 TO FINISHED ( 02 )
001070     ELSE IF ITEM1 (AA) > 2
001080             MOVE 1 TO FINISHED ( 02 )
001090             STOP RUN
001100     ELSE SET AA UP BY 1.
001110 SEARCH03.
001120     IF A > 10
001130             MOVE 1 TO FINISHED ( 03 )
001140     ELSE IF B (A) = 4
001150             MOVE 1 TO FINISHED ( 03 )
001160             STOP RUN
001170     ELSE SET A UP BY 1.
001180 SEARCH04.
001190     IF C > 2
001200             MOVE 1 TO FINISHED ( 04 )
001210     ELSE IF I = 2
001220             MOVE 1 TO FINISHED ( 04 )
001230             STOP RUN
001240     ELSE IF I = 4
001250             MOVE 1 TO FINISHED ( 04 )
001260             GO TO PARA-A
001270     ELSE SET C UP BY 1.
001280 PERFORM05.
001290     IF X EQUALS 9
001300             MOVE 1 TO FINISHED ( 05 )
001310             ELSE PERFORM PARA-A THRU PARA-B
001320             ADD 2 TO X.

```

DESIGN AND IMPLEMENTATION OF A
GENERAL PURPOSE MACROPROCESSOR FOR
SOFTWARE CONVERSION

by

DAVID A. SCHMIDT

B. A., Fort Hays Kansas State College, Hays, Kansas, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

In order to facilitate the conversion of approximately 100,000 lines of IBM AHS COBOL to a version acceptable to the current COBOL compiler of the Interdata 8/32, a preprocessor will be implemented. This device will automatically reconcile approximately ninety percent of the needed conversions by number; the manpower effort saved is predicted to be considerably more.

The device as implemented is to be as conducive to change in terms of host and target languages as is possible; this facilitates the preprocessor's later use in applications nonrelated to the current conversion effort and allows the device to change as the details and problems of the conversion become clearer. The preprocessor will make use of instructions which are commonly called macros. These macros will be stored in a tabular format that can be easily altered to suit the needs of the user.

Accompanying the preprocessor will be a manual of documentation describing the mechanics, implementation, and use of the device. This manual is to be so written that others may be able to use it as a basis for further modification, extension, and application of the work accomplished.