DOCUMENT RESUME

ED 167 122 IR 006 868

AUTHOR	Gardner, Edward
TITLE	Programming Language CAMIL II: Implementation and Evaluation.
INSTITUTION	Air Force Human Resources Lab., Lowry AFB, Colo. Technical Training Div.
SPONS AGENCY	Air Force Human Resources Lab., Brooks AFB, Texas.
REPORT NO	AFHRL-TR-78-45
PUB DATE	Aug 78
NOTE	67p.; Appendixes may be marginally legible due to light and broken type
AVAILABLE FROM	Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402 (671-056/76)
EDRS PRICE	MF-$0.83 HC-$3.50 Plus Postage.
DESCRIPTORS	*Computer Assisted Instruction; *Computer Managed Instruction; Computer Programs; Cost Effectiveness; *Instructional Improvement; Instructional Systems; Military Training; Program Evaluation; *Programing Languages
IDENTIFIERS	*Computer Software; PASCAL

ABSTRACT

A reimplementation of Computer assisted/managed instruction language (CAMIL) for qualitative and quantitative improvements in the software is presented. The reformatted language is described narratively, and major components of the system software are indicated and discussed. Authoring aids and imbedded support facilities are also described, and key CAMIL programs used in the development are discussed. The resulting system offers a method for future improvement of the Air Force Advanced Instructional System (AIS) computer support system without expenditure of additional funds for computer support. (Author)

AFHRL-TR-78-45

AIR FORCE HUMAN RESOURCES LABORATORY

PROGRAMMING LANGUAGE CAMIL II: IMPLEMENTATION AND EVALUATION

By

Edward Gardner

TECHNICAL TRAINING DIVISION
Lowry Air Force Base, Colorado 80230

August 1978

Final Report for Period February 1977 - May 1978

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235

NOTICE

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related Government procurement operation, the Government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This final report was submitted by Technical Training Division, Air Force Human Resources Laboratory, Lowry Air Force Base, Colorado 80230, under project 2313, with HQ Air Force Human Resources Laboratory (AFSC), Brooks Air Force Base, Texas 78235.

This report has been reviewed and cleared for open publication and/or public release by the appropriate Office of Information (OI) in accordance with AFR 190-17 and DoDD 5230.9. There is no objection to unlimited distribution of this report to the public at large, or by DDC to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved for publication.

MARTY R. ROCKWAY, Technical Director
Technical Training Division

RONALD W. TERRY, Colonel, USAF
Commander

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFHRL-TR-78-45	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* PROGRAMMING LANGUAGE CAMIL II: IMPLEMENTATION AND EVALUATION		5. TYPE OF REPORT & PERIOD COVERED Final February 1977 – May 1978 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Edward Gardner		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Technical Training Division Air Force Human Resources Laboratory Lowry Air Force Base, Colorado 80230		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2313T407
11. CONTROLLING OFFICE NAME AND ADDRESS HQ Air Force Human Resources Laboratory (AFSC) Brooks Air Force Base, Texas 78235		12. REPORT DATE August 1978 13. NUMBER OF PAGES 66
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

languages	computer workload	high-level language
CAMIL language	computer programs	computer software
PASCAL language	program performance	structured programming
computer services	computer assisted instruction	programming language conpilers
computers	computer managed instruction	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A reimplementation of Computer assisted/managed instruction language (CAMIL) for qualitative and quantitative improvements in the software is presented. The reformatted language is described narratively, and major components of the system software are indicated and discussed. Authoring aids and imbedded support facilities are also described, and key CAMIL programs used in the development are discussed. The resulting system offers a method for future improvement of the Air Force Advanced Instructional System (AIS) computer support system without expenditure of additional funds for computer support.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

SUMMARY

Objective

The system software in the Air Force Advanced Instructional System (AIS), while providing necessary classroom support for courses at Lowry AFB, did not meet original design performance objectives. In addition, due to cost and other impacts, full features of the Computer assisted/managed instruction language (CAMIL) implementation were not realized in the initial implementation. The objective of this work unit was to determine whether a different approach to the implementation of CAMIL could meet original performance objectives and also implement the full language and authoring aid system while simultaneously offering improved maintainability.

Approach

The CAMIL language was slightly modified to improve compilability and program readability. A new compiler for the language was implemented, based upon top-down recursive analysis rather than the table-driven approach used in the original compiler. The system support program was rewritten in a high level language, and the system was configured to run with a reduced level of interaction with the operating system. Several service functions were transferred to peripheral processor routines to allow for greater parallel processing, and key CAMIL programs were rewritten using the new system. The resulting system was to be performance compared with the original system in detail, but this has been deferred due to a change in operational requirements.

Results

Over 95% of the system has been implemented in the high level language PASCAL for ease of maintenance of the system software. The new compiler runs approximately 10 times faster than the original, and several possibilities remain for further speed enhancement. The new system provides for an elaborate group of authoring aid functions while imposing no additional burden upon the author, and numerous further programming aids could be added to the new configuration. The resulting CAMIL programs appear to run from 5 to 20 times faster than their predecessors, but this relationship has not been rigorously tested as was originally intended.

Conclusions

A path for considerable qualitative and quantitative improvement in the AIS system software is available if and when system loading increases due to demand for AIS computer services.

PREFACE

We would like to acknowledge the support of the AIS computer operators who helped us during the long nights when this work had to be done. We would also like to thank Harold Montgomery of the McDonnell Douglas Corporation for his help in understanding the internal operation of the existing AIS computer operating system. We specially thank Lt Col Roger Grossel for his support in initiating this work unit and for his faith in our abilities to improve a highly complex system with the limited manpower and resources available in our organization.

TABLE OF CONTENTS

Table of Contents (*Continued*)

LIST OF ILLUSTRATIONS

PROGRAMMING LANGUAGE CAMIL II: IMPLEMENTATION AND EVALUATION

I. INTRODUCTION

The language described in this document has been implemented in support of a large scale effort within the United States Air Force training community to apply computer technology to improve technical training efficiency. The major effort in this program has been to apply individually assigned self-paced learning methods to four high-student-load training courses at an Air Force technical training center. Within this environment, a large scale computer has been programmed to manage the instructional programs of approximately 2,400 students by tracking their performance and capabilities and assigning appropriate instructional packages based upon their past and predicted performance. The computer also performs many of the administrative tasks created in such an environment, keeping all student records necessary to properly manage each student individually. One of the available instructional media will be interactive computer assisted instruction (CAI), also supported by the central computer.

In order to implement the above software, the implementation of a contemporary programming language capable of servicing both student management and student instructional terminals, as well as software development, was deemed necessary. Before the decision was made to develop a new language and/or implementation, current languages supporting similar activities were reviewed. After determining that such an integrated attempt at computer assisted/managed instruction had never before been attempted on the scale of this effort, it was also determined that suitable software had not been previously developed in support of such an application. The most closely related efforts were a large scale computer managed instruction (CMI) system at the Naval Air Station in Memphis, Tennessee, and the Plato IV effort at the Computer Based Education Research Laboratory of the University of Illinois in Champaign-Urbana. Although both were outstanding examples of their respective types of programs, it was felt that neither offered software capable of supporting the type of integrated CAI/CMI environment being sought. For these reasons, it was decided that a contemporary programming language supporting the best current programming practices would be specified and implemented to support the number of students anticipated in the projected Air Force training environment. This language was identified as CAMIL, a mnemonic for Computer Assisted/Managed Instruction Language.

Because both the original implementation and the one described in this report are referred to as CAMIL, the two languages have been referred to as CAMIL I and CAMIL II. This report will for purposes of brevity use the term CAMIL for the second implementation since our purpose is primarily to describe it rather than to compare the two implementations. In the few places in which the two are being compared, suitable discrimination will be made.

CAMIL can be described in customary terms as a high level, general purpose, interactively implemented, ALGOL-like, extensible programming language. The syntactic format of the language is generally like that of ALGOL; while the semantic features of the language generally represent extension and generalization of the facilities of current PASCAL. A major addition to its capability is the inclusion of an English-like statement called a "sentence" composed from "words" such as "verbs," "prepositions," and "adverbs." New words may be defined within the program, effectively allowing new statements to be added to the language, within a predefined flexible syntactic format. Another major facility added, which also supports sentences, is the support of multi-element expressions or groups of values. Such tuples may appear as lists of verb objects in sentences, or as values which may be assigned to multi-element, user declared types such as arrays or records. The user may also declare new prefix, infix, or postfix operators for existing or user defined types, or may extend existing operators to new user defined types. The language also includes a large standard library of defined sentence words allowing highly self-documenting programs to be implemented by relatively unskilled programmers.

CAMIL is compiled into absolute binary code for the Control Data Corporation (CDC) CYBER 70 series computers. The compiler is written in PASCAL and implements a process called intelligent partial compilation. All CAMIL programs are interactively edited by an on-line modular editor written in CAMIL, which cooperatively structures CAMIL programs for modular compilation and leaves information for the compiler to use in avoiding unnecessary compilation of unchanged modules. Likewise, the compiler generates and stores cross-reference information which it uses to determine rippling effects of editing changes in order to cause recompilation of affected modules. Using this technique, it is not unusual to recompile a 5000-line program in several central processing unit (CPU) and real-time seconds since much input/output (I/O) and processing can be avoided in a typical compilation situation.

In order to facilitate analysis of student data and generation of periodic reports in batch mode, CAMIL has been implemented using the same addressing conventions as PASCAL, thus allowing compatible descriptions of data collected on line to be analyzed with PASCAL programs even though packed records or arrays may exist in the CAMIL data base. An interface package allows any batch program to call out the same disk I/O services available in CAMIL to access the student data base accumulated by CAMIL programs running in real time. In addition the CAMIL system allows programs to be detached from their initiating terminal and run in a "background" mode at a service limited priority; this provides for data analysis and processing in CAMIL without necessarily reserving a computer terminal. To facilitate general usability of the system, the CAMIL compiler and PASCAL compiler both interface directly to the CAMIL data base so that rapid turnarounds of compilations can be achieved without using the system printer, thus allowing any terminal in the system network to be used for software development. This authoring environment constitutes a very important part of the CAMIL authoring system and has a direct impact on the productivity of the CAMIL programmer.

In order to keep the compiler and language description closely related, the flow chart descriptive method developed by Wirth in reference 1 will be used to describe the CAMIL grammar. CAMIL has been designed for rapid compilation with as few forward references required as possible. Procedures need not be forward declared, and labels need not be declared at all. The simple (although semantically powerful) syntax results in raw compile speeds of about 250–300 lines/sec within declarations and about 100 lines/sec within executable statements on the CYBER 173/6 (Control Data 6400). With the partial compilation technique mentioned above, this often results in effective compilation rates greater than 1000 lines/second. Error recovery is particularly good since the compiler can abort compilation in almost any module it gets "lost" and continue to other modules without the sometimes devastating effect caused by mismatched symbols such as parentheses or BEGIN/END pairs.

To clarify some of the examples included in this paper we must briefly explain the hardware environment in which CAMIL executes. CAMIL is implemented in a 96k central memory CYBER 73/6 processor and services a current network of 80 Magnavox Plasma Display terminals through a digital television communications network originally designed at the Computer Based Education Research Laboratory of the University of Illinois (Reference 2). The network also includes ten "intelligent" student management terminals (optical forms reader, printer, PDP 11/05) which use a compatible protocol on the same communications hardware. The network is expandible to over 1000 terminals within this basic hardware. All displays presented at these terminals must be produced by CAMIL programs. Two CYBER control points (i.e., partitions, jobs, etc.) are serviced at high system priority to provide synchronous data transfer (program DRIVER, executed each 1/60 of a second) and interactive execution of CAMIL programs (program EXECUTER, voluntarily releasing the CPU only when CAMIL requests are satisfied). The initial CAMIL implementation currently services approximately 2700 military students, primarily through CMI services at student management terminals, while interactive terminals are currently used primarily for data base management, software development, and materials authoring. The language described in this document represents a more advanced version of the language based upon the earlier experience and was intended to improve the language qualitatively while offering major improvements in implementation efficiency and authoring turnaround time.

Report Organization

This report has two major subjects: the first encompassing the language and the second encompassing the software elements needed to implement the language upon the CYBER computer. We have tried to present a narrative description of the language and implementation, rather than a formal language reference manual, in order to impart to the reader an understanding of the effort required to implement this type of software and of how the language and implementation relate to several other contemporary languages and implementations. What we have found most difficult to place into words has been the impact of the interactive and dynamic authoring environment implemented by this system upon ourselves as programmers. We have viewed this project from the beginning as the construction of a motivating and enabling tool for programmers and course developers which would allow the rapid development and evaluation of interactive computer assisted instruction and management. Although the potential of such an environment has not yet been demonstrated, we now have the ability to make such an environment available.

Language Description

This section of the report will provide a general description of the CAMIL language. It is intended for a reader who has a working familiarity with contemporary high level programming languages, such as ALGOL, PASCAL, or JOVIAL. This reader should easily recognize the purpose for including most of the described features in a language such as CAMIL; therefore, very familiar data types or statements are not described in great detail. Constructs unique to CAMIL are described narratively in greater detail so that the reader will be able to relate these to facilities which might be represented by other constructs in other languages or which might not be available in other languages.

The CAMIL syntax is described by a set of grammar charts using the basic style used by Wirth (Reference 1) to describe the syntax of PASCAL. No formal production or reduction grammar exists for the CAMIL II implementation, reflecting the fact that a top-down, recursive descent compiler is used to implement the language. A reduction grammar for the original CAMIL implementation contained about 400 productions, some of which correspond to features which have not yet been implemented in the original system; this indicates the complexity which may be anticipated when a table driven, bottom-up compiler is used to implement a complex language, such as CAMIL. In comparison, the syntax chart description for the current CAMIL is very compact and is readily related to the compiler structure for maintenance purposes, although it does not have the guaranteed relationship that a reduction grammar has with respect to the compiler. In the case of CAMIL II, the grammar does correspond to a language in which ambiguity can be resolved by looking ahead one token at most.

Within this notation, an oval box ( ) is always used to surround a reserved word in the language; such words are built from upper case letters and are intercepted by a bottom-up lexical scanner and classified as single symbols. Reserved words and punctuation are also occasionally surrounded by a small pointed box ( ), which is pointed in the direction of production flow of the grammar and which is identical in meaning to the symbol ( ). A strictly rectangular box is used to contain the name of another chart, with the implication that some compiler routine will be recursively called to collect the item implied by the name of the box ( ). The lines with arrowheads indicate the direction of production flow of the charts. To relate this notation to a more familiar form, the examples below indicate equivalent Backus Naur Form and syntax chart examples.

Backus Naur Form

```
<constant decl> ::= CONSTANT <typespec list>
<typespec list> ::= <typespec list> ; <typespec pair>
                  | <typespec pair>
<typespec pair> ::= <typespec> <decl pair list>
<decl pair list> ::= <decl pair list>, <decl pair>
                   | <decl pair>
<decl pair> ::= <identifier> = <constant expression>
```

Equivalent Syntax Chart:

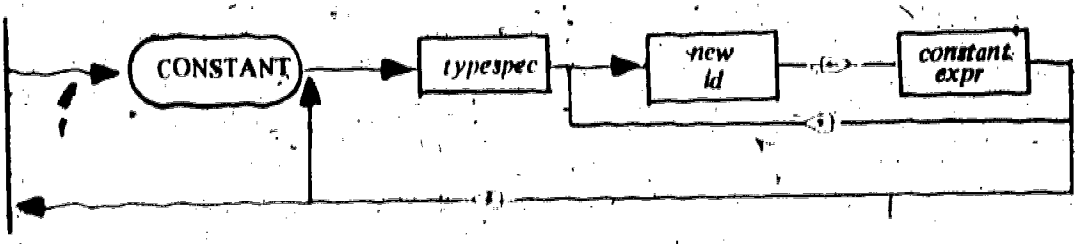


Figure 1. **Example Syntax Chart.**

The full syntax chart for CAMIL has been reduced to two pages and this represents a considerably more compact and understandable description than an equivalent reduction grammar. Since the subtleties of language semantics are more difficult to present in an organized and pictorial manner than language syntax, examples will be included in certain sections of this report to indicate how language structures have been used to implement built-in facilities, many of which are actually coded in CAMIL.

II. CAMIL LANGUAGE OVERVIEW

Most languages require that a program be described syntactically such that key or reserved words indicate the major divisions of the program structure. In CAMIL, a program is always entered, edited, and executed on-line. All programs are stored in a direct access file system with a set of directory elements describing the modules representing the CAMIL program. Major sections of the program are represented by separate directory chains. A program directory entry is used to link to the sets of modules which comprise the CAMIL program; all directory entries are automatically created and deleted as the user adds modules to or deletes modules from the program. Any CAMIL program consists of one or more of the following types of modules.

SHARED data --data global in scope but addressable by every executing copy of this program
PRIVATE data --global in scope but a separate allocation is kept for each executing copy of this program
PROCEDURES --recursively callable subroutines
SEGMENTS --sections of code which can be branched to/from any segment or procedure and which normally constitute major parts or sections of the CAMIL program

CAMIL contains only three definitional levels: one level for predeclared, built-in data and subroutines, a global level of data accessible from any segment or procedure, and a local level of data within any procedure or segment. Only one segment may be executing at any time, but there is no specific limit to the number of procedures which may be recursively activated at any time. While this is more restricted than PASCAL, it substantially simplifies many author and user problems which might otherwise arise when asynchronous features of the language are used. (These are explained later.)

SHARED and PRIVATE modules are used only to contain type and storage declarations and cannot be executed. Execution of a CAMIL program begins at the first line of the first segment in the program directory. After the last line of a segment is executed, control transfers to the first line of the following segment; the program ends when it is specifically exited by an exit function or when the last line of the last segment is executed. Control may also be transferred to any segment by a "GOTO" to the name of the chosen segment.

Procedures are not executed unless specifically called and will return control to the statement following their call unless a "GOTO" to some segment is executed within the procedure. An escape to any segment results in an escape from all procedures currently activated and a return of all local storage allocated for the currently active procedures and segment. CAMIL allows the program author to also define asynchronously active "function keys" in the program which will transfer control at the user's initiative to places designated by the author. Thus, while the execution sequence of a program is determined by strict rules of the language, the actual path taken through a program can be as flexible as the program author wishes to allow.

CAMIL programs can communicate with each other through several possible means. Two different executions of the same program can transfer data through SHARED data modules or through the CAMIL data base which is a record oriented, direct access file system. Several different CAMIL programs can also communicate through another type of SHARED data module called SYSTEM SHARED which must be defined at the system level as part of a special CAMIL program containing the shared data modules. Any two authors can also communicate in real time through a direct message facility between terminals. Other facilities allow a program author to execute a program in small steps, interactively autopsy a program on request, monitor the display of a student who is executing his program, and trap execution errors, along with the complete data situation at the point of failure.

III. CAMIL LANGUAGE DESCRIPTION

Program Structure

CAMIL consists of two distinct but merged parts, the core language and the extensible language. The core language is generally compiled directly into machine code which implements its meaning, the extensible part is mainly implemented through built-in or user defined procedures, which supply semantics to sentences and extended operators. All syntax is fixed, but within the extensible part of the language, it is rather flexible. The core language supports basic types such as integer, number, character, logical, string, and textual displays. From these basic types, more complex types, possibly containing multiple components, may be defined by the user.

CAMIL makes a distinction between INTEGER, which may be used as numeric or as bit information, and NUMBER, which is assumed to be a real number and subject to the usual side-effects of truncated precision machine arithmetic. CAMIL is tolerant in its conversions between these types and also when it compares internal numbers to responses entered by humans, who are generally less precise than floating point arithmetic units. CAMIL supports a 252-character set, 126 of which are permanently fonted on the terminals and system line printer and 126 of which can be fonted within the interactive terminals within an 8 by 16 dot raster pattern as the author desires. Strings are allowed over the full 252-character set, and a special construct called a "wordstring" can be two-dimensional, allowing a complete screen of data to be written with a single write sentence.

CAMIL supports RECORD and ARRAY structured types and also allows the author to specify that they be PACKED insofar as reasonable for data space conservation. Arrays are normally indexed by INTEGER expressions or user defined ranges of mnemonic values as in PASCAL. In CAMIL, however, the CASE variant of PASCAL is generalized, any field may be a variant field, and the case selector field is automatically set by generated code whenever a record is composed as a multi-element expression. This important distinction allows the compiler to pass type data to the executing CAMIL program and will be explained in conjunction with the sentence extensibility feature which it supports. Unlike PASCAL, CAMIL allows multi-element literals to be composed, thus allowing ARRAY and RECORD expressions rather than forcing the user to explicitly assign each field of a record or an array. This is particularly important when combined with other aspects such as OPTIONAL fields in a record, it allows a user to define sentences completely in CAMIL which are substantially more complex than the typical read or write statement and which may be written with an arbitrary number of parameters and modifiers. CAMIL also maintains definitional identity between composed expressions and the actual parameter lists of procedures and also between procedures, formal parameter lists, and record definitions. Thus a procedure can be considered as an operator defined upon a record definition, and a procedure call can be considered as a prefix operator acting on a composed expression. Infix and postfix operators are an immediate extension of this idea which provides a uniform basis for operator extensibility implementation.

CAMIL supports a direct access data base through several simple file operators. All files are shared among all CAMIL programs and may be opened simultaneously by any CAMIL programs permitted access by a file security system. Programs are by name permitted to perform specific file operations on designated

files. Operations supported allow individual records to be read, written, deleted, or updated. Records are automatically reserved while being updated to avoid the problem of two different executions or programs updating the same record. Files may be accessed by index or sequentially or by direct address. All files are structured as files of some specified type. File identifiers can refer to either the current data contained in the central memory file buffer or to the associated file sequence stored on the disk, depending upon whether the context in which the identifier appears implies a reference to data or a file operation. A single statement similar to the PASCAL WITH statement allows a particular record to be reserved, readup, dereferenced as in the PASCAL WITH, updated, rewritten, and released. All file operation statements allow an ELSE clause which is processed instead of the file operation in the event that the file operation cannot be successfully completed. Although maximum file size is specified when a file is defined, the actual number of records in the file is dynamic and the presence or absence of a particular record can be determined.

The familiar IF-THEN-ELSE statement is supported by CAMIL, and an additional statement is added to support asynchronous interruption of the normal program flow by the user if programmed by the author. The author can use IF-DO statements to provide an asynchronous transfer of control to the "DO" statement in the event that the user presses one of the function keys listed in the "IF" clause. This feature allows the author to make a great number of options available to a user without having to check explicitly for them at any time. Certain "built-in" conditions may also be handled using this feature, such as file errors, system termination by the operator, and processing errors.

The WITH statement from PASCAL is implemented, as is the ubiquitous GOTO statement. A form of the GOTO statement is provided which combines the GOTO and CASE statement functions. In this GOTO CASE form, the selector expression transfers control to a selected tagged statement, but branch instructions are not generated at the end of each case. This results in a behavior similar to the computed GOTO statement while retaining the structural form of the CASE statement and achieves the semantic efficiency which in certain situations the computed GOTO provided. CAMIL also extends the CASE statement to include an ELSE clause which allows a closure to the set of possible values of the selector. The familiar FOR, REPEAT, and WHILE statements of PASCAL are combined into a single iterative statement allowing optional selection of any or all of the above possibilities and also the BY increment, somehow lost in the transition from ALGOL to PASCAL. An iterative case like statement, called the JUDGE statement, is allowed and provides for the collection of an input from the user, the comparison of that input with possible matching anticipated answers, execution of a consequent in the event of a match, or execution of an optional ELSE closure condition if no match is found, followed by resolicitation of the response when no match is found. Many different possibilities of action are easily specified by the author due to the flexibility of the response accepting sentence. The JUDGE statement is highly usable in many situations in which responses are solicited from student users and was derived from the TUTOR language (Reference 3). The RETURN statement has also returned and provides a needed alternative to the labels which otherwise crop up on the last statement of a procedure in those cases where structured programming does not quite suffice to express an algorithm.

An important feature of the CAMIL design is based upon a type of CAMIL statement called a sentence. The syntax for the sentence is built on several parts of speech commonly used in simple English imperative sentences. The syntax allows the author to rearrange the parts of a sentence in a manner which makes semantic sense in English. In this format, verbs, adverbs, prepositions, and objects (expressions) can be rearranged in the manner most convenient to the user without affecting the meaning of the sentence. Thus a sentence such as

```
"write x on line 10,col 5 for 5 sec"
```

would execute exactly the same if it were written

```
"for 5 sec write x on line 10,col 5"
```

just as it would have the same apparent meaning to a human observer reading both forms. The user can add new verbs, adverbs, prepositions, and also operators, which function as adjectives in appearance, by adding procedures which implement the meaning of these words and define acceptable combinations of verbs and prepositional phrases. In the CAMIL implementation, terminal hardware dependent I/O is predefined within this facility during compiler initialization, thus removing I/O from the core language and providing highly readable I/O statements. It is hoped that this type of facility may offer a workable solution to the problem of authoring readable programs in languages which must be tailored to meet the needs of particular equipment.

Procedures and functions may also be called using conventional parameter lists. No restrictions are placed on the size of the objects returned by functions in order to allow support of arbitrary user defined types as function results. Operators defined by the user are treated as functions of one (prefix, postfix) or two (infix) operands and produce a value usable in any context in which a computable expression is allowed. The execution of a sentence is an activation of the procedure of definition for the verb of the sentence and executes as efficiently as any normal procedure.

The operation of assignment is fully implemented since it has been extended to include literals of any type. Additionally, the user may explicitly "cheat" between size compatible types by "casting" an expression as another type. This usually machine dependent, sometimes regrettably necessary operation, can thus be clearly indicated in the CAMIL program and implemented with minimal overhead. The resulting expression provides explicit notice of what must be reexamined if the program is moved to a different CAMIL implementation.

The notion of a NAME as an attribute of a variable or record field allows a uniform treatment of this concept within the language. A NAME field within a record or parameter list is conceptually identical to a NAME variable in a normal data area. The normal assignment operator is made transparent when NAME identifiers are used, since the assumption is always made that the referent of an identifier is always intended when an identifier is used, unless otherwise specified. Thus names need not be dereferenced explicitly as in PASCAL. An additional arrow operator "->" is implemented with the meaning "x->y: make x point to what y is pointing to". In this manner, a NAME parameter to a procedure is treated exactly as a NAME global variable or as a NAME field within a record. Storage may be dynamically allocated within a program execution through the use of a MAKE operator which allocates data in an area with a lifetime corresponding to the lifetime of the pointer with which it is affiliated.

CAMIL implements compile time resolution of constant expressions which reduces the size of program code and allows computations to be introduced into constant initializations. This has hidden benefits in that constants such as "2/3" or 1.5x10↑5 may be stated accurately at compile time yet appear in familiar notation to the user. It also allows PACKED constant composed expressions used for initializations to be packed at compile time, thus avoiding generation of the codes necessary to do this which are usually larger than the resulting expression. The following operators are available in general between the indicated types of operands:

Arithmetic	Addition, subtraction, real division, integer division, integer remainder, exponentiation, and negation, these are defined between INTEGER and NUMBER operands and returning INTEGER and NUMBER results.
Logical	Union, intersection, difference, word shifts in the left and right direction with zero pad and end around carry forms, and bitwise complement of words, these are defined between INTEGERS and produce an INTEGER result.
Set	Union, intersection, difference, and complement, these are defined between compatible sets.
String	Concatenation and infix search between STRING operands and between STRING and CHAR operands.
Relational	Equality and inequality between compatible types and relational operators between pairs of most types and set membership. Pointer identity between NAME operands, as well as normal equality between their referents.

Conversions: Upon assignment between all reasonable combinations of basic types.

User defined: Any operators definable between any kind of operands if the relationship is definable using the above operators upon the components of the user defined types.

Data Declarations

All data accessed in CAMIL must be named and typed. These declarations fall into four basic classes: TYPE, CONSTANT, VARIABLE, and NAME.

A TYPE declaration is merely a convenient way of associating a complicated data description with an identifier so that the identifier may be substituted for the more complicated definition without typographical error. Either a TYPE identifier or an explicit data description may be used whenever a "typespec" is indicated by the CAMIL grammar. A typespec must be associated with any data used by the program, and the compiler will check to insure that only semantically meaningful operations are attempted between data items according to their type.

A CONSTANT declaration associates an identifier with a typespec and with an initialized, unchanging value. Constant identifiers may be used anywhere in place of the value with which they are associated, but their value cannot be changed during the execution of the program. Their permanent, unchanging value must be stated in their declaration.

A VARIABLE declaration also associates an identifier with a typespec and a storage allocation which can contain an object of the indicated type. An initial value may be indicated for the variable as part of the declaration. The time of allocation of the storage is the time at which the variable is initialized, thus the following initialization times hold for the indicated class of variables:

SHARED	When the first program referring to the shared module is loaded
PRIVATE	Each time a new user begins to execute the program (even though someone else may already be executing the same program)
PROCEDURE	At each activation of the procedure
SEGMENT	Whenever the segment is activated

If no initial value is specified for a variable, the associated storage will be cleared (zeroed) at the time of activation.

A NAME declaration associates an identifier with a typespec and a pointer which can only point to an object of the indicated type. An occurrence of a name variable in any context causes the name to be dereferenced to the corresponding object. Storage is allocated for the pointer when a declaration is encountered, but not for an object of the indicated type (these are created dynamically) and no initial values are allowed for names. Name variables are initialized to NIL references at the times indicated above for variables.

Declaration syntax is independent of the type of module in which the declaration occurs and is indicated in the CAMIL syntax diagrams included in the Appendix. The following examples were excerpted from the CAMIL program editor and are offered without semantic explanation at this time as samples of data declarations.

Examples:

```
CONSTANT
  INTEGER buffer_size←255, maxmods←237;
  ARRAY [0:10] OF STRING[11] modtypes←
  ("Comments","Shared Data","Private Data","Procedures",
   "Segments","Job Cards","Main Block","Input Data",
   "Text","Macros","Functions");
```

```
TYPE
 0:buffer_size BUFFERRANGE;
 0:maxmods MODRANGE;
 0:63 IDCHARS;
 0:2↑15-1 DISKADDRESS;
 0:2↑13-1 JULIANTYPE;
 0:2↑16-1 ADDRRANGE;
 PACKED ARRAY[10] OF IDCHARS PACKEDNAME;
 PACKED RECORD
  BEGIN
   PACKEDNAME modulename;
   DISKADDRESS src,obj,lv;
   MODRANGE headingend;
   JULIANTYPE updtdate;
   1:240 cellnumber;
   ADDRRANGE srcsize,objsize,lvsize,baseaddr,ownaddr
  END MORECORD;

VARIABLE
 INTEGER
  grid_start←1, grid_spacing←5, nbr_grid_lines;
 LOGICAL
  insert,recompile,
  inspect_only←TRUE;
 PACKED ARRAY[32] OF 0:33 screenlines←(1,2,3,4,5, 27*0);

NAME
 MORECORD current_module_directory;
```

Data Definitions

Language Tokens

The tokens from which a program can be composed fall into traditional categories. These basic elements are: reserved words, identifiers, literals, and punctuation.

Reserved Words

The following list of upper case spelled reserved identifiers are identified by the CAMIL compiler as built-in delimiter tokens in the language. They cannot be redefined by the author, thus they will always have the same meaning in any CAMIL program. The role of these words is to clarify the structure of the program to the compiler and to the original and subsequent authors of the program. The list is presented at this time for reference. The words are reserved in UPPER CASE only, but some of them also appear in the language as predefined identifiers in lower case.

IF	DO	OF	BY
TO	END	FOR	SET
OWN	THEN	ELSE	CASE
FROM	WITH	GOTO	TYPE
NAME	VERB	PREP	FILE
BEGIN	ARRAY	WHILE	UNTIL
JUDGE	REPEAT	PACKED	RECORD
RETURN	SWAPPED	VARIABLE	CONSTANT
OPTIONAL	PROCEDURE		

Identifiers

An identifier is defined as a group of upper or lower case letters, digits, or underscores. The first character must be an upper or lower case letter. The compiler only attaches significance to the first 10 of these characters and ignores any additional ones. An identifier must appear on a single line, i.e., an end of line signals an end of identifier to the compiler. Identifiers are used to name data items, modules, and locations (labels) within modules.

Punctuation

Punctuations are used in CAMIL as separators for the purpose of program clarity and as operators or grouping symbols. The following general uses are described for punctuations:

```
{}          Braces enclosing comments ignored by compiler
()          Parentheses used to group elements of expressions
[]          Square brackets used to enclose index expressions
             and literal sets
;           Semicolon used to separate statements
:           Colon used to visually separate items such as
             labels and statements, or to denote an optional
             data item in certain sentences
┌┐          Shorthand versions of the reserved words
             "BEGIN" and "END". These characters display
             on the terminal screen as corners ┌ and └
             and are automatically connected by the program
             editor with a vertical line which serves to
             emphasize the nesting structure of the program
             while encouraging neatly paired indentation
← →         Assignment operators
=≠■≤≥<>¬∨∧  Logical operators
+-×/÷↑ //   Math operators
⊂∪∩∈        Set operators
‖\          String operators
" "         Quotes used to delimit screen messages
' '         Quotes used to delimit character strings
.           Operator used to denote reference to
             fields of records
π           Synonymous with the constant 3.141592654
*           An operator used in multivalued expressions to
             denote that a particular value is to be
             repeated within the expression
.           Decimal point used in expression of decimal
             fraction
#           Used to identify a hexadecimal constant
@           Used to identify an octal constant
_?'!&%      Several other pieces of punctuation commonly
             used in English, but not assigned any special
             syntactic meaning in the CAMIL language
```

Detailed uses of punctuations are shown in the CAMIL syntax charts included in the appendix.

Literals

The CAMIL language provides for the representation of literals, i.e., self-defining constants of all basic data types supported by the language. The types and formats of these items are:

```
LOGICAL:    TRUE  FALSE
INTEGER:    ddddddd          {digits 0..9}
            @ddddddddd       {octal digits 0..7}
            #ddddddddd       {hexadecimal digits 0..9,A..F}
NUMBER:     .ddddddddd
            ddd.dddddd       {digits 0..9 max of 10 sig digits}
            ddddddd.
CHAR:       'c'              {a single character in quotes}
STRING:     'ccccccccc'      {0 to 120 characters in quotes}
POINTER:    NIL              {means an undefined referent}
WORDSTRINGS:
          "The rain in Spain falls
           mainly on the
           plain"            {used to display data on terminal}
```

The above constants are limited in accuracy corresponding to the accuracy of the AIS computer by the following rules:

1. No integer may be defined with a precision of information denoting more than 60 bits of binary data. The compiler limits octal constants to 20 digits and right justifies fewer than 20 digits in a field of zeroes. The compiler limits hex constants to 15 digits and right justifies fewer than 15 digits in a field of zeroes. The compiler limits decimal integers to the largest value which will fit within 45 bits of information since this is the precision of the AIS computer multiplier; the value of this largest integer is 2↑45-1.
2. No number may be defined with more than 10 digits of decimal precision. While this is less than the AIS computer provides, it is consistent with the accuracy obtainable after repeated arithmetic operations of functions. No number expression may appear as a constant or be computed to exceed approximately 10↑295. Numbers may be expressed in scientific notation as constant expressions in the formats used to describe expressions as explained in later sections,e.g., 25.4x10↑15.
3. Wordstrings are two-dimensional chunks of character information used to place information on the display screen of an AIS terminal. Any characters except the double quote " may be used in the wordstring. If a wordstring is broken across more than one line, the first word of the next line will be left justified against the left margin in effect for the terminal when the message is written (leading blanks are ignored in lines of a wordstring). There is no specific limit to the size of a wordstring.

CAMIL supports data types of the above literals through operators and through facilities for compounding the above types into aggregates or indexable groups. On simple scalar types it also allows the limitation of attention to subranges of these types. The operations between these types will be explained in the section on expressions; the grouping mechanisms will now be explained.

Record Grouping

CAMIL allows certain values to be grouped together and optionally compressed for minimum storage utilization. This grouping of heterogeneous item types is called a record or packed record. The definition of a record must inlcude names for all of the fields within the record and indicate the type of each field. Variations within a portion of a record are allowed when the contents of the record might be used to represent more than one kind of thing through a type of field called a variant. A variant selector field is associated with a variant field to designate which alternative is in effect at any time. The record definition is often associated with a type identifier in a type declaration to avoid the possibility of erroneously repeating the definition and to save space within compiler tables. The syntax chart is shown in the appendix, but an example is included here to clarify the intent of the record declaration.

Example:

```
PACKED RECORD
 BEGIN
  0:999999999 ssan;
  0:9999 squadron_numbe:, student_dorm, student_room;
  (AB,A1C,A2C,SSGT,TSGT,MSGT) student_rank;
  15:65 student_age;
  (MALE,FEMALE) student_sex;
  CASE LOGICAL transient
   BEGIN
    TRUE:"0:999 next_base; DATE_TYPE out_process_date";
    FALSE:"STRING(30) permanent_organization";
   END;
 END STUDENT_RECORD
```

Array Grouping

CAMIL also allows homogeneous types of data to be grouped into an indexable array structure. The array may also be compressed for minimum storage utilization by including the word PACKED in the array definition. An array definition must include the range of indexes allowed for each dimension of the array and must also designate the type of elements which are being grouped together. The range is denoted by including a subrange of the indexing type in the definition or by denoting the largest index and allowing the compiler to generate a default minimum index of "1". Syntax for the definition is included in the appendix, and several examples are included below:

Examples:

```
ARRAY[10] OF INTEGER;
PACKED ARRAY[10:20] OF NUMBER;
PACKED ARRAY[1:10, 1:15, 1:20] OF LOGICAL;
PACKED ARRAY[15]OF
  PACKED ARRAY[4] OF
    PACKED RECORD "INTEGER i,j,k; NUMBER n";
```

File Grouping

A file may be declared in a CAMIL program in order to gain access to data in the CAMIL data base. The purpose of the file definition is to associate some specific data file by name with a variable in the program which is capable of holding an element of the file. The file identifier thus defined is regarded by CAMIL as both the name of the variable and the name used to refer to the data base file during some file operation. The file declaration must name the data base file, the type of item which the program considers to be in the file, and the program name of the variable which contains a record from the file. References to the data contained within the file variable are obtained by simply mentioning the name of the file variable as explained in the description of the file statement. The syntax for the file definition is included in the appendix but an example is indicated below.

Example:

```
FILE "studentdata" OF STUDENT_RECORD
```

Expressions

The CAMIL expression mechanism provides for all of the normal types of expressions and operators found in most high-level languages, such as PASCAL, but also provides many extended features which other languages do not have. Some of the extended features include special set operators, multi-element expressions (composed expressions), a type casting mechanism, and user defined operators.

The lowest level of precedence encompasses the relational operators "=", "≠", "<", ">", "≤", "≥", "≡", "∈". The meanings of the first six operators are similar to other programming languages. The "≡" operator is used in two different ways within CAMIL. The most common use is to determine whether two pointers have the same referent (i.e., the addresses denoted by the pointers are equal). The other use, while similar in appearance, is to determine whether an optional record field or procedure parameter is present in a record or procedure call. This is tested by comparing the name of the parameter to a NIL pointer. The form of this test is "parmname ≡ NIL" and it returns a true value when the parameter does not exist. The "∈" operator is the "contained in" operator used to test set membership. The test "5 ∈ s" would be true if 5 is a member of the set s.

The next highest precedence level contains the "+" and "-" prefix operators and also the "+", "-", "v", "u", "@", "‖", "←←", "→→", "→↑", "↑←" infix operators. The "+" and "-" will not be discussed, since their meaning should be clear. The "v" operator is the logical "or" operator. This operator has two logical operands and returns a logical result, which is the inclusive-or of the operands. The "u" operator is the set union operator. This operator computes the union of two set operands. The exclusive-or operator "@" is also a set operator which computes the logical difference of two operands. The operators "←←" "→→" "→↑" "↑←" are used to shift integer operands. The "←←" and "→→" operators are left and right end off shifts, respectively, with zero padding. The "→↑" is a right circular shift, and "↑←" is a left circular shift. The "‖" is the string concatenation operator. This operator merges two string operands producing a single string as a result.

At the next precedence level, the "x", "/", "÷", "Λ", "n", "//" operators are found. The "x" and "/" are the normal multiply and divide operators, "÷" and "//" are integer divide and remainder operators. The logical "and" operator is "Λ", and "n" is the set intersection operator.

The next precedence level contains the "↑" operator and user defined postfix and infix operators. The "↑" operator is the power operator, which can have integer or number operands. To express 2 to the n^{th} power "2 ↑ n" would be used. The user defined postfix and infix operators which are referenced by an identifier are also found at this precedence level. An example of this type of operator is the postfix "sec" operator, which looks like "5 sec" when used.

The highest precedence level includes the operators "¬" and "N" and also parenthesized expressions, cast expressions, user defined prefix operators, and composed expressions. The "¬" operator is the logical "not" operator, and "N" is the set complement operator. Parenthesized expressions have the standard meaning that the expression inside the parenthesis is evaluated prior to using the entire parenthesized expression as a result. Cast expressions allow one type of expression to be considered (or cast) as another type of expression by placing a "type id :" in front of it. This is a very useful feature to have in a typed language such as CAMIL because there are many times it is desirable to override the typing conventions of the language (especially in system programs). A simple example (assume c to be of type character) "INTEGER : c" allows the internal value of c to be used as an integer. Records and arrays can also be type cast, allowing multi-access methods to the same storage area. User defined prefix operators which are recognized by identifiers, such as "line" and "col", are also at this precedence level.

Composed expressions are also at the highest precedence level. When a composed expression is encountered, the composing routine is passed the type of the object to be composed so that each element in the expression list can be added to the stack in the proper location. The resulting multi-element item in the stack can then be used as the object of a verb, the parameter of a procedure call, as a value to store into some variable, or as some value which is part of another composed expression. Because the type of the object being composed is known, full syntactic and semantic error checking occurs as the expression is scanned. The following is an example of a simple composed expression which assigns a value to the write cursor AT, which is a record with two integer fields "AT ← (5,8);". Values can be repeated in a composed expression by using the "★" repeat operator. This is especially useful when initializing an array and many of the elements are to have the same values. For example, if a is a 10-element array of integers, the following will initialize the first element to 1, the second element to 5, the last element to 84, and the rest of the elements to 15 "a ← (1,5,7★15,84);".

The extensible features of the CAMIL language allow a user to declare prefix, infix, and postfix operators. There are two methods available to the user for declaring operators. One method is to declare an operator which is to be identified in the program by an identifier (the "line", "col", and "sec" operators are defined in this way). When an operator is declared in this manner, the operator takes on a precedence depending on whether it is a prefix, postfix, or infix operator. The prefix operators "line" and "col" are in the highest precedence level while "sec", a postfix operator, is at the next precedence level. The other type of operator declaration is one in which an existing operator is extended to new operand types. The user extended operator acquires the precedence of the operator symbol it is extending. A familiar example is the extension of the common arithmetic operators to include complex operand types. The "+" operator will be extended in the following example. The definition of the type COMPLEX is:

```
RECORD
 BEGIN
  NUMBER real,imaginary
 END
COMPLEX;
```

The following is a procedure heading which defines addition (using the + symbol) of two complex numbers:

```
COMPLEX + (COMPLEX a) + (COMPLEX b);
```

Assuming the name of the module is plus, the procedure body for the above procedure heading is:

```
BEGIN
 plus.real ← a.real + b.real;
 plus.imaginary ← a.imaginary + b.imaginary;
END;
```

Whenever two complex numbers are to be added in the program, the + operator will invoke the defining operator procedure. If a,b,c are all declared as complex numbers, then the statement "c ← a + (b + c) + b" would be possible. An entire set of such operators can be defined over complex numbers and stored in a library, which a user could reference whenever he wished to perform computations using complex numbers. A new number system could also easily be implemented by extending existing operators to operands in the new system. The extensibility which CAMIL offers is quite adequate for many different and interesting applications.

Constant expressions, including packed composed expressions, are evaluated at compile time and the resulting values are stored in the program. The use of constant expressions reduces the program object size because the expression is not computed at run time. This feature also allows the programmer to change storage allocations throughout a program by changing a few simple constants used in other constant expressions. This enhances the maintainability of programs by allowing objects, such as tables, arrays, and lists to be rapidly and uniformly modified throughout a program.

Executable Statements

CAMIL provides a large group of permanently defined statement types for the construction of algorithms. Most of these statements are defined with reserved words as delimiters and several include one or more other imbedded statements. Many also represent verbatim equivalents of standard statements from ALGOL and PASCAL, as would be expected. Several others represent generality extensions of existing types of statements, and several are somewhat new as far as we know.

Old Favorites

Compound Statement

CAMIL provides for the grouping of several statements to produce one single apparent statement through the familiar BEGIN-END pair of delimiters. It also provides single character equivalents of these

through two characters "⌐" and "¬". These often reduce program text size making it possible to place more program on a display screen by allowing several statements to be placed on a single line. CAMIL also frequently uses BEGIN END pairs as grouping elements in data declarations and in CASE and JUDGE statements. All BEGIN END pairs are fully matched, unlike in PASCAL where END sometimes appears without a corresponding BEGIN.

Labeled Statement

CAMIL allows identifier labels to precede statements for purposes of branching to the statement with a GOTO statement. Labels need not be declared and may be forward referenced within a module. The scope of definition of a label is the module in which it appears. The name of any segment type module is also considered to be a label to which control may be transferred by a GOTO statement.

RETURN Statement

A RETURN statement, meaningful only in a procedure type module, is provided and is equivalent to a GOTO to a label following the last executable statement in the module.

IF-THEN-ELSE Statement

A traditional branching statement is provided with the usual meaning of executing the statement following THEN, if the expression following IF is evaluated true, and executing the statement following ELSE (if present) if the expression following IF is false.

GOTO Statement

The familiar but wisely avoided GOTO statement is also provided for use in escaping to segment type modules or for transferring to local labels within a module. Transfer is allowed to any point within a module which can be labeled, so the GOTO may be fully exploited and abused.

WITH Statement

The PASCAL WITH statement has also been implemented to allow local dereferencing of record names. The effect is to make any field of the dereferenced record usable as a simple identifier within the statement to which the WITH prefix is attached, just as in PASCAL. A variation which allows a file record to be read up, dereferenced, updated, and replaced is also implemented and explained in a later section.

The other statements in CAMIL have either been developed or originated, and they will be explained more fully in the following sections.

Modified or Improved Statement Forms

Assignment Statement

The assignment statement is present in its familiar form for left assignment. Compared to PASCAL, this statement has been extended by extending the types of expressions which are allowed in the language. Since CAMIL allows the user to structure multi-element data types through ARRAY and RECORD declarations, it also allows the user to compose expressions for this type of operand. The resulting "composed expression" can be assigned as a value to a variable of the record or array type. Examples would be:

```
PACKED ARRAY[1:10]OF INTEGER i;
i←(3, 18, 5, 27, j+6, 9, i[3], 3*22);

RECORD
 BEGIN
  INTEGER k;
  NUMBER m,n,p;
  CHAR c;
 END rec;
rec←(29, 30.5, 40.8, 27 + 4*10↑27, "W");
```

23

CAMIL also provides a form of assignment which allows NAME variables (pointers) to be assigned the value of other pointers. The normal meaning of left assignment in CAMIL is "copy the storage associated with the right operand into the storage associated with the left operand". Normal variables will have their address assigned at compile time and the compiler will generate code to copy the required amount of storage using these known addresses. If either or both of the variables are NAME type variables, the address of the storage to be copied will be taken from the pointer whose address is known at compile time. Thus normal left assignment always involves the REFERENT of the indicated identifier or expression. The address held by a pointer may be copied into a compatible pointer by using a right arrow "x←y" operator with the associated meaning "make x point to the same address in storage that y is pointing to". Since the attribute of being a NAME belongs to an identifier rather than to a type as in PASCAL, pointers always dereference to things rather than to other pointers, thus the up arrow used in PASCAL to denote whether a pointer is being dereferenced is not needed. The nicest part of this definition is that it makes the meaning of the left assignment uniform across constants, variables, and names while still allowing pointers to be used in the more unusual cases in which they must be dealt with as addresses to be copied. We feel that the resulting syntax is more concise than the syntax in PASCAL, as is shown in the following example

```
PASCAL:
  Value copy:   xt := yt
  Pointer copy: x := y

CAMIL:
  Value copy:   x ← y
  Pointer copy: x ← y
```

The comparison becomes more significant as the expression becomes more complicated, as in the following four-step pointer dereferences where the pointers are fields within records

```
PASCAL:   isdt.idtypedt.fstfieldt.idtypedt.size.wds := wds;
          isdt.idtypedt.fstfieldt.idtypedt.size.bits := bits;

CAMIL:    isd.idtype.fstfield.idtype.size ← (wds,bits);
```

The PASCAL statements in the above examples accomplish the same results as the CAMIL statements beneath them but are more prone to errors since some fields require dereference with arrows and some do not. The composed expression used in the CAMIL assignment further points out the desirability of assigning values to multi-element data items. The example is taken from the CAMIL compiler.

Function Key Statement (IF-DO Statement)

The CAMIL language provides for the support of a set of keys and conditions which may occur asynchronously during the execution of a CAMIL program. The author of a CAMIL program frequently wishes to provide for the whims of the person executing the program in case he wants to change his mind, to back out of a situation he has entered into inadvisably, or to seek at variously predictable times the help of the program author. In effect he needs to provide standing offers to process such requests and to link them to tangible things which the user can do to request the actions. Some systems provide for this with a command mode or control key escape. CAMIL provides this through support of a set of 30 function keys on the user keyboard and several pseudo "keys" which can be "pressed" by the system when some situation the author may wish to process has occurred.

The author denotes a set of conditions, which he is willing to process, at a place in the program where he wishes to process the conditions. If one of the activated conditions is encountered, control will be transferred to the "statement" to which the activated set of conditions is attached and continue thereafter to the following statements appearing after the IF-DO statement. The author may redefine the place and the conditions by inserting a new IF-DO statement in the program in the execution path. The format of the statement is

```
IF  setofconditions  DO  statement ;
```

The set of conditions is an expression and is calculated and activated when the IF-DO statement is encountered in normal program execution sequence. At the time of normal encounter, the "statement" following DO is not executed. It will only be executed if one of the conditions in the condition set occurs. The statement following DO is often a CASE statement with tags for each of the activated conditions. The following example shows how the IF-DO statement is used to provide help to the user, a way out to the previous module, and time left to answer the question.

Example:

```
IF (HELP,BACK,F1,GOINGDOWN) DO
 CASE SYS.FKEY  (A system variable tells which key pressed)
  BEGIN
   HELP: write "At this point you may only answer the
                question as asked, press BACK to return
                to the previous example, or press the
                F1 key to see how much time you have
                left to answer the question" for 5 sec
                on line 27, col 5;
   BACK: GOTO module27;
   GOINGDOWN: GOTO system_crash;
   F1  : timeleft←45-SYS.TIMER-STARTTIME;
         write timeleft on line 30; GOTO redo_question;
  END;

erase; STARTTIME←SYS.TIMER; timeleft←45;
on line 10, col 5 write
"What is the moment of the force you have specified
 when it is applied on a 20 ft moment arm";

redo_question:
 accept with [digits# for timeleft  ec;

 etc.
```

In the above example the GOINGDOWN condition is one which is asserted after the operator requests CAMIL to terminate its operations 30 seconds prior to its actually stopping all CAMIL program activity. This author has decided to use the condition to transfer control to a module which might store the student's current status on disk so that he can be restarted after the system resumes operation. Since the condition might occur at any time, even while the program is waiting on the user to answer the question, the program must be asynchronously primed to deal with this eventuality.

IF-DO condition sets are "stacked" in a nested manner when procedures are called and conditions apply at the most recent level at which they are active. Thus a HELP key might be active in a segment and in a procedure currently being called by the segment. If the HELP key is pressed in this situation, the HELP condition in the procedure would be processed rather than the one in the segment. If the BACK key was defined in the segment but not defined in the procedure and was pressed while the procedure was active, control would be transferred back to the segment to the statement following the DO of the IF-DO activating the BACK key. The program stack would of course be adjusted so that it corresponded to the state in which the segment was active but the procedure had not been called.

File Operations

File operations have been integrated into CAMIL syntax to provide for file requests to manipulate data on the file and to also use the file name as the name of a buffer containing one element of the file. All

files are declared in the program as a file of a certain type of element. The file identifier is then a variable of the type of the element. The CAMIL file manager allows three basic kinds of files: indexing, direct, and variable. The meaning of these terms will now be defined as they pertain to CAMIL files.

Indexing files are files accessed by a file key, i.e., a piece of data which is used to discriminate between different records. In the design of the CAMIL system, it is intended that indexing files will be used as control and directory functions and not used to store large records of data. In the implementation of the file manager, indexing files are locked into the executive control system (ECS) of the CYBER but protected on disk whenever they are written. The result of this approach is that READ access to indexing files may always be accomplished without physical I/O delay or time slicing of the program. The cost is that records cannot be very large without wasting a large quantity of ECS.

Direct access files are files of fixed record size accessed by a specific record address. Records may be buffered (more than one logical record within a physical record), and new records may be written into empty records automatically allocated by the file manager. This access method is implemented so that the disk address of each record may be computed from the record address, so that only one physical file operation is required to obtain the desired record. Direct access addresses range from 1 to the number of records which may occur in the file.

Variable size files are implemented so that there is no limit on the size of records other than the limit stated in the file definition. Also, with variable size files, only a single physical access is required to obtain the desired record. Space is allocated for these records so that the minimum number of disk sectors needed to contain the size of record written are allocated. Actual record size is maintained by the file manager and any record update which changes the size of the record written results in automatic reallocation of disk space to accommodate the additional size. All disk space is automatically recovered when records are deleted, and "checkerboarding" of available sectors is prevented by consolidation of adjacent sectors.

The following variations of four basic commands handle all file operations permitted on CAMIL files under control of a CAMIL or batch program attached to the CAMIL data base.

Examples

```
READ file()                    (Reset to first record)
READ file                      (Sequential read)
READ file(index)               (Read particular record)
READ file(index)tsize          (Read variable size record)
WRITE file                     (Sequential write)
WRITE file(index)              (Write new(index=0)or old rec)
WRITE file(index)tsize         (Write variable size record)
DELETE file                    (Sequential delete)
DELETE file(index)             (Delete particular record)
DELETE file(ALL)               (Purge contents of file)
WITH FILE file(index) DO       (Reserve and read denoted file)
                               (record, dereference fields if)
                               (RECORD type, replace updated)
                               (record, release reservation)
```

A file index is optional with the meaning of sequential access to the next record in the file if it is omitted. The record size identifier is only used for variable sized records and is automatically set to record size when reading and controls the size of the record being written. The ELSE statement is optional and is only executed when the file operation cannot be fully completed. A built-in system variable contains the actual file error which has occurred and may be interrogated for use in deciding how to process the error. Any file statement which does not return fully successful caused NO alteration to the file on disk. A file may be reset to the first record for sequential processing by reading with an empty index.

The name of a file type variable has a dual role in CAMIL. When the file name is preceded by a file operator, such as "READ", an operation upon the disk file associated with the file name in the file declaration is performed. Within any other usage context, the file name is a variable of the type of data indicated in the file declaration. If the file is a NAME type variable, a pointer is associated with the file name. Like any other pointer, this pointer is the address of storage allocated to contain an object of the type of the file. In this manner, a single disk file name can be used to read into several buffers, some of which may be dynamically created and only used locally in a procedure for example.

The operations supported by the file manager include specific functions for each type of file request, which is in turn dependent upon what fields are included or omitted in the file statement and the type of file the action is performed upon. The user sees a much simpler interface, since he is only presented the operators READ, WRITE, DELETE, and PURGE and a record-updating construct based on the WITH statement. The READ function only reads records into the file buffer. The DELETE function deletes the designated record from the file. The PURGE function deletes every record from the file. The WRITE function replaces the designated record on the file if it is found or adds it to the file if it is not present. A construct is provided with the syntax:

```
WITH FILE myfile(index) DO <statement> ELSE <statement>
```

When the above construct is executed, the designated record is checked to insure that it is not reserved by some other program. The record is then reserved for this program, read into the file buffer, the statement following the DO is executed, the record is rewritten onto disk, and the reservation is removed. The reservation step assures that two programs do not read the same record, update its contents, and then rewrite it oblivious to the fact that each has updated the same record. The ELSE clause is executed only if the record cannot be reserved and/or read. Several attempts to reserve the record are automatically initiated by the file manager to eliminate the need for the programmer to handle the rare case when another program might hit the same record. The WITH FILE form also accomplishes the same function as the normal WITH statement as it dereferences the designated record in the same manner described above for the WITH statement.

File security is performed by the file manager. When a file is defined in the CAMIL data base, it is potentially available to every CAMIL program. Access is controlled in the definition process (an interactively run CAMIL program) by allowing the file definer to enable specific file manager functions for any CAMIL program by name. Since these names are unique, the file manager can thus authorize specific programs to perhaps read a file, but not add to it or alter any records in the file. Provisions are also made to control batch job file security by associating permitted file operations with the CAMIL program from which they are submitted. By configuring the system so that requests can only come through the CAMIL system and its associated peripheral processor routine, the CAMIL data base is fully protected from direct invasion, and access must come through the file security process of the file manager. The file editor can also define default security entries to allow files to be accessed by programs without specific security access if it so desires.

Iterative Statement

The CAMIL iterative statement combines in a single statement all of the functions of all three PASCAL/ALGOL iterative statements. The statement is based on the following reserved words, all of which are optional:

FOR	Followed by a variable which is initialized when the loop is started and incremented by the value of the BY expression as the loop repeats
FROM	If present, denotes the starting value of the FOR variable; defaults to 1
TO	If present, denotes the stopping value beyond which the FOR variable will not continue; defaults to the largest integer
BY	If present, denotes the increment to be added to the FOR variable each time through the loop; defaults to 1

REPEAT If present, denotes the maximum number of times the loop will repeat, regardless of other control mechanisms; defaults to a large implementation dependent number

UNTIL If present, is followed by a logical expression which will be evaluated at the end of each loop and which will terminate the loop if the value is TRUE; defaults to FALSE

WHILE If present, is followed by a logical expression which will be evaluated at the beginning of each loop and which will terminate the loop if the value is FALSE; defaults to TRUE

If none of the optional words are present, the loop will be terminated only by intentional exit or by reaching the implementation dependent default value of the REPEAT phrase. All repeat computations involving the phrases FOR, FROM, TO, BY, and REPEAT are resolved by the compiler or generated code prior to loop initiation and result in a maximum iteration limit. This computed value controls iteration along with the UNTIL and WHILE expressions. For this reason, assignments to the FOR variable within the loop will affect the values it assumes, but will not affect the number of iterations in the loop.

The following simple example shows the advantage of combining loop functions into a single statement:

```
CAMIL:
  FOR I FROM 10 TO 1 BY -2 UNTIL FM.ERR=EOF DO
   "READ myfile; array[I]=myfile"

PASCAL:
  I:=10;
  FOR J:=1 TO 5 DO
   BEGIN
    READ myfile;
    IF EOF(myfile) THEN GOTO 15
    ELSE
     BEGIN array[I]:=myfile^; I:=I-2 END;
   END;
  15:
```

The CAMIL form is not only an improvement in flexibility for the programmer, but the routine used to compile it is smaller than the three routines used to compile the three PASCAL iterative statements.

Case Statement and GOTO Case Statement

The CAMIL case statement is a simple extension of the ALGOL/PASCAL case statement. The extension adds an ELSE clause for logical completion of the set of possibilities. In the event that one of the designated tags is not found to match the CASE selector expression, the ELSE clause will be executed if present. This ability eliminates the frequent need in PASCAL to imbed the CASE statement in an IF-THEN-ELSE statement which can be particularly awkward if the chosen tags cannot be expressed as a set. It also results in a more efficient implementation of this rather frequent concept while clarifying the intent of such a combination.

A new form of CASE, called the GOTO CASE, is added. In this form, which appears identical to the normal case statement, the compiler avoids generation of the branch instructions which normally follow the code for each tagged statement. The result is that if control is transferred to one of the tagged statements rather than to the ELSE statement, it and all of the following tagged statements will be executed in turn. The ELSE statement will, however, be avoided. This form of the case provides a direct equivalent for the FORTRAN computed GOTO while giving it the structured appearance of the case statement and avoiding the manufacture of numerous labels to capture this type of logic. While we do not expect to see this form frequently used, it does provide a translational equivalent for the FORTRAN/ALGOL form of computed GOTO and the implementation cost is very small. This statement is represented by preceding a normal CASE statement with the reserved word "GOTO".

Judge Statement

Since the primary implementation context of CAMIL includes the operation of interactive terminals, we felt a strong need to include a specific statement for the acceptance and evaluation of responses. After observing the implementations of many systems, it was determined that one of the most powerful response acceptance mechanisms was implemented in the TUTOR language (Reference 3). The essence of this mechanism in CAMIL is a combination of an accepting, processing, evaluating, and looping function combined into a single statement. The statement is called the JUDGE statement and is so named after the JUDGE contingency structure implemented in TUTOR. In CAMIL, the JUDGE statement has the following syntactic form:

```
JUDGE
 <response accepting sentence>
BEGIN
 <expression list> | <action statement>;
 <expression list> | <action statement>;
  ...
 <expression list> | <action statement>;
END
ELSE <no match statement>
```

The response accepting statement is usually an accept sentence acquiring input from the keyboard into a built-in variable called the judge buffer. It can of course be a compound statement which "massages" the content of the judge buffer after accepting the input from the student. Since the normal accept sentence allows many options restricting the input, this statement is not normally needed but it is available.

The expression tag lists are normally anticipated responses or ranges of responses separated by commas. In this way, more than one answer can be associated with each action. Ranges of numbers, integers, sets, strings, characters, and expressions are allowed as tags. The compiler generates logic to convert the contents of the judge buffer to all of the types of things on the included tags, and tries to match each of the tag expressions to the converted content of the judge buffer. If a match is found, the corresponding action statement is executed, and further matching is terminated. If no match is found, the ELSE statement is executed if present.

After the above has occured, a semantic flag is next tested. This flag will have been set true if any match were found and false if no match occurred. If the flag is true when tested, the JUDGE statement will terminate and the following statement will be executed. If the flag is false, control will be transferred back to the accepting statement if a maximum count has not been exceeded. Since some tags might correspond to anticipated "wrong" answers which would require further input, the semantic flag can be reset in the action statement to cause further looping. In the same sense, the flag can be set in the ELSE statement if no further processing is desired. the loop count is also a built-in variable and defaults to no limit if not set by the author. The actual number of times that the JUDGE loops is stored in another built-in loop counter and is available to the author for his use if he needs it. An example follows:

```
erase;
on line 5, col 5 write
"Press the indicated keys to choose a game program.
 Some of the games may not be working yet.  You may
 type in 'quit' if you want to leave now
```

a The Hangman Game
b The Spelling Game
c The Race Game

```
d    The Startrek Game
e    The Spirogram Machine
f    The Empire Game";

JUDGE
 accept on line 9, col 15 with [nocaps,erase_echo]
BEGIN
 'a'!       GOTO HANGMAN;
 'b'!       GOTO SPELLGAME;
 'c':'f'!   for 5 sec write flashing "not working yet";
             J_FLAG←FALSE";
 'quit'!    KILLPGM;
END
ELSE
 for 3 sec on line 30, col 7 write
 "No, enter one of the letters in the menu or type 'quit'
  to leave this program";
```

Sentence Library

Several standard sentences are available in the CAMIL language. These sentences allow the user to perform several needed functions, and some elaborate special purpose functions. Most of the standard sentences are implemented in CAMIL, but a few are implemented in PASCAL to avoid the swapping overhead of frequently used sentences. Before describing the sentences, we shall explore some of the standard prepositions which can be used with the sentences. Prepositions can fall anywhere within the sentence and in any order, as long as they do not interfere with the verb-object phrase of the sentence.

The until preposition has a function key set as its argument and can be used with several verbs. Its function is to provide a set of function keys which act as an until condition of the sentence. When the until condition is reached, the sentence completes execution and program control continues. All of the function keys in the until set are considered as next conditions, and will not be considered as asynchronous function keypresses if pressed when the sentence is in execution. Some examples of the until preposition follow.

The first example is a simple accept sentence in which the programmer wants the user input to be accepted when either a NEXT, BLUE_NEXT, or a HELP key is pressed. When one of the until condition keys is pressed, whatever input the user has entered will be accepted and the program will continue execution after the accept sentence.

```
accept until [NEXT,BLUE_NEXT,HELP];
```

The second example is one in which the until preposition is used with the write sentence. In this case the write sentence will output the information to the terminal and then pause until the until condition is met (the NEXT or BLUE_NEXT key pressed). When the until condition is met, the write sentence will then erase the displayed information and the program will resume control following the write sentence

```
until [NEXT,BLUE_NEXT]
write "NEXT  to continue
       BLUE_NEXT for more information";
```

Once control returns from the sentence using an until preposition, the programmer can find which of the until keys was pressed; in the same manner he can query which function key is pressed in an IF DO statement. In the above example one might want to branch, depending on which key the user pressed (NEXT or BLUE_NEXT). The SYS.FKEY variable contains the desired information so that the program can perform the desired function, depending on the key pressed.

In all of the sentences in which the until preposition is defined, there is also a time limit which may be imposed using the for preposition. When a time limit is imposed, the sentence will pause for the desired amount of time, and if another action has not restarted the sentence when the time limit is reached, the sentence will continue. If both the until and for prepositions are used in the same sentence, the sentence continues execution when either one of the until keys is pressed or the time limit is reached. To indicate units of time in seconds, the postfix operator "sec" is available to make readable time caluses.

A few examples of the for preposition follow. They are similar to the examples for the until preposition except that execution of the program is now resumed after the desired time limit is reached.

```
accept for 5 sec;

write "hello" for 3 sec;
```

There are two prepositions which can be used to indicate screen positions: at and on. The at preposition has two integer parameters whcih indicate actual x and y dot co-ordinates on the screen. The on preposition also indicates a screen location, but on a character level using the line and column operators. Several simple uses of the at and on prepositions follow:

```
accept at 5,10;
accept on line 25, col 2;
write "Next to continue" at 26,15;
write "Next to continue" on line 28, col 15;
```

Other special purpose prepositions are available and will be discussed with their associated verb phrases.

The standard sentence to request input from the user is the accept sentence. There are many variations upon the basic facility for response input. The basic accept sentence automatically places the prompting arrow at the accept cursor, awaits a user response, and erases the prompting arrow when the accepting state is completed. An elaborate sentence example could be one which sets the accept cursor, displays the prompting arrow, limits the input length to three characters, only accepts octal digits, converts the input and stores it into an integer variable i, and places a 5-second time limit on the user's response time. The following example would perform the functions described:

```
accept i:3 OCT for 5 sec on line 24, col 45;
```

To describe the functions of the accept sentence the possible prepositions and defaults will be described. The accept verb has four preposition types; an until set (the until preposition), a time type (for), a screen position (at or on), and a with set (with).

The until preposition temporarily removes the asynchronous nature of the function keys contained in the until set, replacing their meaning with that of an end-of-input terminator. This allows the program to accept inputs and perform different functions on the input, depending on which key was used to terminate the input stream. If no until clause is present in the accept sentence, the NEXT key along with any keys in the system defined variable SYS.UNTIL set are assumed to be the end-of-input keys.

An accept statement may also be terminated by a time limit which is specified in the "for" clause. If a for clause is present, the accept period will be terminated at the end of the specified time limit, assuming the user has not pressed one of the defined until keys. If no for clause is present, the accept period will only terminate when one of the until keys or an active function key is pressed by the user.

The screen location clauses at and on are used to designate where on the screen the accept prompting arrow is to appear. If no screen location is given, the accept will occur just to the right of the last item written onto the screen.

The with clause contains a set which allows several special functions to be performed during the accept state. The functions which can be present in the with set are:

noarrow	accept without displaying the prompting arrow.
noecho	do not echo user input.
nook	do not display the ok/no in a judge statement.
underline	draw a underline displaying the length of the accept limit.
erase_echo	erase the previous input in a judging state.
allcaps	translate all alphabetic keys into capitalize mode.
nocaps	translate all alphabetic characters into lower case mode.
touch	activate the touch panel and accept data from it.
digits	only echo/accept digits 0-9 and symbols ".", "+", and "-".
octaldigits	only echo/accept octal digits, 0-7 and "+", "-".

The accept verb also has an optional object which may be the subject of the accept. If an object is present, the accept verb will convert the input to the type of the object and store the results into the accept object. For example, if "s" is of type STRING, then "accept s" will place the user input into the variable "s".

When using an accept object, the accepting limit and accepting action limit can also be specified. The accepting limit indicates the maximum number of characters which will be echoed/accepted. Any characters which are pressed after the accept limit is reached (also available through J.LIMIT) are ignored. The accept action limit (J.ACTION) causes the NEXT key to be pressed when the indicated number of characters have been input. Thus an accept with the accept limit set to 1 will immediately continue execution after one character is input by the user. The method of indicating the accepting limits is by placing a ": accept limit : action limit" after the object. Thus accept 1:5 places an accept limit of 5 characters on the accept, and accept 1:5:5 places an accept and action limit of 5 on the accept.

The accept sentence also performs the necessary conversions to the type of the accept object. For example, if accepting into a type Boolean, the accept sentence will convert the input string TRUE into the internal representation for the Boolean value of TRUE. As an added feature, when accepting into an integer or number, the accept sentence will automatically specify an all digits accept condition so that only digits are echoed. It is also possible to signify an octaldigits condition by placing ": OCT" after the accept object, or just OCT after an accept or action limit.

The accept sentence also has two alternate forms: accept more and accept rep. The accept more sentence is used to continue accepting starting where the last accept took place. For instance, if the characters abc were accepted and an accept more was executed, the accept cursor would fall after the c of abc, and the abc would be part of the current accept. That means all of the editing keys and erase keys

could be used on the abc just as if it had been typed during the current accept. The accept more sentence does not have a clause for setting where the accept is to occur (obviously since the previous accept is being continued), and it does not have an object for the accept either.

The accept rep sentence is for accepting and representatively echoing user responses. When an accept rep is executed, the "J.REPECHO" flag is set. This informs the driver to return control to the program after one keypress has been received (so the program can provide a response) and also that the keypress should not be echoed. All of the prepositions available with the accept verb are optional items to the accept rep, but no object can be meaningfully accepted into.

The pause sentence.

To temporarily pause program execution, the pause sentence can be used. The pause verb has two optional clauses and no object. There is an optional until clause which signifies which keys can end the pause condition and an optional for clause which can place a time limit on the pausing. If no until clause is present, the NEXT key and any keys defined in "SYS.UNTILSET" are used as the continuation keys. If no time limit is placed on the pause, program execution will be suspended until a continuation key or an active function key, is pressed. Some examples follow:

```
pause;                          {Pause until NEXT is pressed}
pause until [NEXT,BACK];{Pause until NEXT or BACK pressed}
pause for 5 sec;                {Pause for 5 seconds or until NEXT}
```

The write sentence.

The standard sentence to display textual information on the screen is the write sentence. There are several forms of the write verb, but the discussion will start first with the simple form which displays text on the screen. The object of the write verb can have any of the standard types (INTEGER, NUMBER, CHAR, STRING, and LOGICAL) or any string contained inside double quotes ("..."). The write verb will convert any of these types into the proper form to be displayed on the screen. More than one object can be listed with each verb by simply listing them after the verb, i.e.,

```
write "The answer is ",ans," and the average is ",avg ;
```

Formatting is accomplished by following the items to be displayed with ": integer", where the integer is the desired length of the item being displayed. This makes it easy to generate lines of data with column alignment, even when numeric items of different magnitudes are involved. For INTEGER values, if the length specified is not long enough to display the entire value, the length is increased so that the entire value can be displayed. To display an integer or numeric value in octal, an OCT can follow the ": integer" or ": OCT" can be used and the value will be displayed in octal digits. When OCT is used, the displayed number will be displayed with leading zeros as blanks. If, however, leading zeros are preferred (as in memory dumps), they can be specified by using ZOCT instead of OCT in the sentence. The following example uses a length limit, OCT, and ZOCT:

```
write 1:3 OCT,  ',memory[1]:10`ZOCT  on line 1,col 5;
```

The precision of NUMBER values is controlled by specifing the entire character length of the number to be displayed, and also the number of digits to display on the right-hand side of the decimal point. The form is similar to PASCAL and looks like ": | : p", where | is the total number of characters to display (including the decimal point) and p is the number of digits to the right of the decimal point. If the ": p" is left out, the number will be displayed without any fractional part. To display a number (n) with nine places to the right of the decimal point and five places to the left "n : 9 : 5" would be used. Numeric values are displayed in scientific notation (i.e. 5.6 x 10↑8) when there is not a precision specified.

Several prepositions are optionally available to augment the capabilities of the write verb. The at and on prepositions are used to direct where the information is to be displayed on the screen. When using one

of these prepositions, the starting position of the message is specified, and the left write margin is set to the specified column position so that any line overflow is aligned below the first line.

The "for" and "until" prepositions are both available with the write verb and since they perform similar functions, they will be discussed together. The function of these prepositions is to determine how long a message remains on the screen, by waiting on a keypress or time limit. When one of the specified conditions is reached, the information displayed by the write sentence will be erased, and the program execution will continue.

The write verb performs all of its textual displaying in the write mode of the terminal. To perform writing in the erase and rewrite modes of the terminal, the unwrite and rewrite verbs are used, respectively. These verbs are used just as the write verb, since the only difference is that they place the plasma panel into different modes.

When the flashing adverb is used with the write verb, the message is flashed on the screen until a NEXT key or one of the until or for conditions is met. The write flashing clause has the same parameters as the write verb, and the only actual difference is that the message will flash on the screen until some condition is met. A simple example follows:

```
write flashing "You have won" for 5 sec;
```

Other adverbs which are used with the write verb are "large" and "unlarge". They perform the write/unwrite functions except that the text is drawn with vectors instead of with dot patterns. This allows the size and rotation of the text to be controlled by the program, providing a means to write out large headings or to label graphs with vertical titles, etc.

Two optional prepositions can be used with the large/unlarge adverbs: rotated and sized. The rotated preposition is used to control the angle at which the data is displayed with a default in the normal horizontal position. The sized preposition provides the means of stating the size of the data compared to normal size. By indicating two sizes (i.e., "sized 5,4"), the x and y sizes can be stated separately, allowing either tall and narrow or short and wide characters. If no size is stated, the characters are normal size characters.

An example of using the write verb follows:

```
write large "DEMO PROGRAM" sized 4 on line 5, col 10;
```

Graphic sentences.

There are several available sentences to produce graphic displays on the terminal. Lines, dots, and circles can be drawn, using the "draw", "connect", "dots", and "circle" sentences. A brief description of each verb follows:

The "draw" sentence can either draw a line or plot a dot. Two prepositions can be used with the draw verb: to and from. Both of the prepositions require two arguments which stand for x,y co-ordinates on the screen. To draw a line, the from clause signifies the starting point, and the to clause signifies the ending point. The sentence "draw from x,y to x+5,y+5" will draw a line from the point on the screen representing x,y to the point x+5,y+5. If the starting point of the line to be drawn is the current write cursor, the from preposition can be left out. The sentence "draw to x+3,y-4" will draw a line from the current write cursor location (x+5,y+5 if the previous sentence was just executed) to the x+3,y-4 screen location.

To draw a sequence of connected lines, the connect verb can be used. The connect sentence draws a sequence of lines, starting at the first pair of points and connecting all of the listed pairs of points following. Thus the sentence "connect x,y, x+5,y+5, x+3,y-4" would produce the same results as the two draw sentences in the previous examples.

Dots can be drawn using the draw verb by leaving out the to clause. The sentence "draw from x,y" will turn on the dot at the x,y location of the screen. A dot can also be turned on by the sentence "draw" which will turn the dot on at the current write cursor. In a similar manner as for the connect verb, a group of dots could be plotted using the dots verb. The dots verb simply plots all of the points listed in its object list. The sentence "dots x,y, x+6,y+4, x+1,Y-10" would plot the three points listed on the screen.

To draw circles on the screen, the "circle" verb is used. Several optional prepositions are available to modify the type of circle which is drawn, but the only required object is the circle radius. A precision parameter is optional. If no precision is present, the circle routine will choose an adequate number of line segments to use in drawing the circle to produce a smooth circle.

Several optional prepositions are available with the circle verb to control the type and place the circle is drawn. The first optional preposition is one of type "screen location" used to denote the center of the circle. Either of the two screen location preposition("at" or "on") can be used to position the circle. If no position is given, the circle will be centered at the current write cursor.

There are two possible clauses to control the period of time the circle remains on the screen by using the "for" or "until" prepositions. These prepositions work in a similar manner as with the write verb.

Other prepositions include the ability to control the eccentricity of the circle by using the "eccentricity" preposition. This preposition allows circles to be elongated along the horizontal or vertical axis, forming eliptical figures. Arcs of circles can be drawn using the "startangle" and "stopangle" prepositions. By specifying these angles, just portions of a circle can be drawn. Zero and 360 degrees are the default values for the start and stop angles, respectively. The last optional preposition provides for drawing dashed circles.

An example of how to use the prepositions follows. The circle is to be of diameter 50 (in dots), precision of 25 line segments, to be erased after 5 seconds or until a next key is pressed, and with eccentricity of 2.5 (elongated vertically).

```
for 5 sec until [NEXT] circle 50,25 eccentricity 2.5;
```

A simpler example draws a dashed half circle of radius 100, which will look like the letter C, only dashed.

```
dashed circle 100 startangle 90 stopangle 270;
```

The uncircle verb is identical to the circle verb except that the uncircle verb erases when the circle verb draws, and it writes when the circle verb erases.

Other sentences.

Some other standard sentences follow:

The echo verb is used in representative echo modes, that is one key is interpreted by the program as a string and placed on the screen in the proper position using the echo verb. The echoed output is also placed in the "J.BUFF" variable so that it can actually be erased or edited using the erase and edit keys if an accept more is executed by the program.

The erase verb is used to erase individual lines or to erase the entire screen. It has only one optional object, a line number which indicates that only one line is to be erased. The default if no line is specified is to erase the entire screen. When only one line is to be erased, the current left and right margins are used in the erase operation so that if the desired line to be erased contains a graph or figure it will not be erased.

To load special characters into the terminal's random access memory (RAM), the LOADRAM procedure is used. LOADRAM has two parameters: (a) a description of the character to load as a Boolean array and (b) the character position at which the character is desired to be loaded. When loading several

characters into the terminal's RAM, care should be taken not to do a full screen erase before all of the characters have been loaded, since a full screen erase ends all output going to the terminal (see the catchup verb).

To operate the slide projector, the slide verb is used. This verb only requires one parameter: the slide position desired. A negative slide position turns the slide projector lamp off, and any positive integer will position the slide projector and turn the slide on.

The external verb is used to place data on the terminal's external output jack. The verb's object is an integer value which is to be exported to the terminal's jack. Up to 50 values can be placed with an external verb.

The catchup verb has no parameters of any kind and is used to pause the program until all output to the terminal has been completed. This is a useful verb when sending data to the terminal's external jack or when loading special characters into the terminal's RAM since a full screen erase would end all output going to the terminal (including the types mentioned), and the output lines to the terminal are relatively slow.

IV. CAMIL COMPILER PROGRAM

Implementation Factors

The implementation of CAMIL consists of a compiler to translate CAMIL programs into executable code, a terminal driver to schedule and interface the system to actual computer terminals, an executer to manage the program swapping and provide implicit language services, and a large group of capabilities written in CAMIL and available as built-in language features or as system level CAMIL programs. Each of these major areas will be described in a separate section of this report for ease of avoidance by the reader who is not interested in all of these aspects.

Narrative Description of the CAMIL Compiler

The CAMIL compiler is a top-down, recursive descent, single-pass, optimizing, machine-code generating, partial compiler. The major sections of the compiler program are:

1. Interface Section
2. Compilation Driver
3. Lexical Scanner
4. Declaration Section
5. Expression Section
6. Statement Section

The interface section of the compiler includes the attachment to the CAMIL data base, the compiler initialization logic, the request reception logic, reinitialization logic to compile more than one program, and logic to perform initial processing of source modules.

The compiler driver includes logic to read up and process all of the necessary modules to determine whether partial compilation is suitable and then to determine which of the source modules must be recompiled. It selectively directs compilation of affected modules and stores the resulting machine code and initialization data as needed.

The lexical scanner consolidates character strings into identifiers, numbers, and strings, and categorizes these elements as to type, returning one element each time it is called.

The declaration section of the compiler scans data and procedure declarations within the source code and builds internal symbol and structure tables for use by the expression and statement sections. It also stores the initialization values for constant and variable data types produced by the expression section.

The expression section of the compiler provides for the evaluation of constant expressions and the generation of code for the computation of computable expressions. It also computes the parameter lists for procedure calls and lists of expressions used as the objects in sentences.

The statement section of the compiler scans and generates code for the execution of CAMIL statements. It calls the expression routine as needed to compile expressions embedded within the executable statements.

Data Base Interface

The compiler interface section is necessary because CAMIL programs are stored in a structured direct access data base. Rather than appearing as a stream of characters as is often done, CAMIL programs appear as blocks of lines of characters. These blocks, or modules, are created by an on-line editing program in which the structure of the program is built incrementally as the modules are entered. The program is structured from a program directory which contains the disk address of module directories. Each of these directories contains the addresses of source, initial values for data, and machine code modules for up to 30 modules of the program. The compiler reads the actual lines of the program by using the address of the source modules to read the source modules from disk. After the module has been compiled, the resulting machine code, if any, is stored on disk, along with an initialization module for any locally declared variables if needed. The addresses of these created modules are then recorded in the module directory page which is rewritten to disk after all entries on the page are compiled. There is also a record containing all intermodule cross-reference sets, and an error module containing the location and type of any syntax errors. Any active autopsy records are also attached to the program directory.

Since the compiler is written in the PASCAL language which provides no interface to CAMIL, the PASCAL compiler has been modified to accept CAMIL file declarations and file access statements. This allows the compiler to read and write records on the data base whenever CAMIL is running on the system, even though the compiler is running at a separate batch control point. Separate CAMIL files are defined as follows:

```
PD:       The file of all program address and status info
PDDATA:   The file containing detailed info about programs
MD:       The file of all module and post mortem directories
ERROR:    The file of all error modules
SOURCE:   The file of all source and post mortem data
OBJECT:   The file of all code and initial values modules
CD:       The file of all partial compile data records
```

The PASCAL file interface automatically opens defined files upon first access and closes them upon compiler completion. The compiler synchronizes actions with the program editor (from which compilation requests are made) by inspecting and changing program status in the PD file. File accesses are made through a group of procedures which centralize all data base access for maintenance purposes and process any I/O errors which may have occurred while accessing the dat base.

Since the CAMIL compiler was designed as a resident compiler, it was intended to be initialized once and then would compile programs upon request indefinitely. Also, since the language definition includes many "built-in" routine libraries and variables for interfacing to the interactive terminal, these must be in the compiler symbol table at the outermost lexical level. The interface routines accomplish this by first creating a request to compile a program which contains these definitions and then establishing the resultant symbol tables at a point where the reinitialization logic will not remove them as it prepares to compile the next requested program. The compiler is returned to this configuration prior to compiling each program.

Because of the modularity of the program, the usual overhead items such as code buffers, line and column counters, etc. must be reset as each module is entered. The interface logic performs each of these tasks and reads source data into the input buffer and initializes the lexical scanner. Due to the partial compilation logic, only modules which have been changed or affected by changes need to be processed, thus saving I/O as well as processing time.

Compilation Driver

The compilation driver activates the major sections of the compiler and decides which modules must be recompiled. The process begins by accepting a compilation request and reinitializing the compiler, which is a very simple step. It then looks at information stored with the program to see if anything has been changed since the last compilation which would force the program to be totally recompiled. Such conditions might be a new version of the compiler or executer, changes to the definitions of built-in data, or compiler failure during the last compilation. If this is not the case, a partial compile is instituted.

The compiler decides what to compile by keeping cross-reference sets for each module of the program. It uses set logic to determine whether editorial changes to definitions of data or procedures will ripple to the executable code modules. This is done by considering direct changes to definitions, changes to definitions used in subsequent module definitions, and changes which affect the addressing of variables in subsequent modules. The program editor assists in this by keeping procedure headings actually separate from procedure bodies, although the editor and listing program disguise this fact from the user. In this manner, it can be noted when the user has changed the heading, thus causing modules which refer to the edited module to also be recompiled. Internal changes do not of course require this and they are by far the most frequent type of change.

By performing quick set union and intersection operations, a compile set of modules is constructed which is then used by the driver as it reads module directories to determine when it should activate the module compiler. During actual module compilation, the symbol table lookup routine enters the number of the module which contains any identifier it has found into a "refers to" set for the module it is compiling. This set will be saved for the partial compilation decision in the next compilation. Naturally, any module which contained errors in the previous compilation must be recompiled, and this is reflected in an error set generated during each compilation, which is also factored into the set logic. Actual compilation steps are activated by calling a module compiler which first compiles declarations and then executable code as appropriate for the module. These are activated as the declaration and statement sections, both of which call the expression section.

Upon completion of compilation of all modules, the compiler then calls appropriate parts of the interface section to store the error and cross-reference data and releases the program for execution or for repair of syntax errors through further editing.

Lexical Scanner

Since the CAMIL compiler is single-pass in design, the lexical scanner is designed to be called by the parsing routines and will return a single token at each call or identify that there are no further tokens in the module. Scanning results are stored in global variables, one of which is available for each primitive type of literal or token that can be built from characters. The token encountered is categorized into a major symbol class which denotes the fundamental type of the token, i.e., particular reserved word, comma, parentheses etc. Some of these fundamental kinds are further classified to provide more detail. For example, a RELOP or relational operator would be further classified into EQUAL, NOTEQUAL, LTHAN, GTHAN, etc. This dual classification allows all major syntactic delimiters to be placed into a single PASCAL set, which is quite important in the error recovery process.

The scanner is designed to work with the information format of the CAMIL editor. The editor removes any leading blanks on source lines and packs the string length in characters and the number of leading blanks into the last two characters positions of the last word of the string. The word size of the string is packed into the rightmost four bits of the first word and last word of the string, which enables both the compiler and editor to identify the size and last word of the string. The strings of source are otherwise treated as part of a large packed array of seven-bit characters and thus the leftmost 56 bits of each word contain eight seven-bit character fields. A single word can thus contain up to a six-character string, while the largest string can contain up to 120 characters. The four-bit word size field contains 0..15

denoting a string length of from 1..16 words. Since the CAMIL character set can denote up to 256 character positions, an escape code (the LANG keyboard key) is used to switch from the permanent 128 characters to the user-loadable 128 characters. The infrequency of this alternation results in good packing for data within the CYBER 60-bit word size.

The scanner is also responsible for constructing the internal representation for textual displays used in screen display sentences. In this case, the text is compressed into a special six-bit format essentially ready for immediate release to the display terminal. In this mode, each line of the textual display will be left-justified against the margins in effect at the time of display, thus achieving a very close relationship between the appearance of the text within the CAMIL program and its appearance on the screen when the program is executed.

The scanner will enforce lexical rules for the composition of literals, such as identifiers, numbers, and strings, and will also enforce semantic restrictions such as the size of numbers, limits on numeric precision, or bit size for octal and hexadecimal constants. Although a character pointer is not maintained explicitly for speed purposes, the current scan position within the source buffer is maintained by the scanner and is used by the error reporting routine to construct the exact column position at which an error was detected. Upon reaching an end-of-module condition, the scanner will return an end-of-module token, and if within a quoted string or similar token, it will produce an appropriate error message. This is needed to handle the occasional error of mismatched quotations or failure to close a comment and allows the compiler to limit the problem to the module in which the error was introduced.

Declaration Compiler

The declaration section of the compiler is activated when procedure declarations are scanned, when private and shared data definition modules are scanned, and at the beginning of each executable module if declarations are present. The results of any of these activations are the creation of tree structured descriptions of any types declared in the program, the construction of the symbol table for identifiers defined in the declarations, and allocation of storage to contain program variables. Because CAMIL constants and variables may be initialized, the declaration section must also construct the run time representation of initialized storage and provide for saving this information.

CAMIL provides no forward procedure declaration. All procedure modules defined on an editor directory page have their headings located together in a single module of source text. The program editor provides a function key to allow the author to edit the heading of a procedure while editing the body module and keeps track of the location of each procedure's heading within the single source module. The declaration processor reads this one source record for every page of procedure delcarations and enters all procedure declarations into symbol tables prior to compiling the body of any procedure. Thus all procedure definitions are processed prior to compiling any procedure references, eliminating the need for forward procedure declaration while performing minimal I/O to obtain this information. Module directories contain the names of segment type modules, and these are also entered into the symbol tables as available labels to which control may be transferred.

The declaration section is next applied to all global level (private,shared) modules. Since the basic format of CAMIL declarations is <type specification> <name list> for any class of storage (constant, variable, name), a common TYPESPEC routine is provided for processing all type definitions while separate routines (VARDEC, CONSTDEC, and NAMEDEC) are provided for processing the differing requirements for each of these storage classes. Because of the limited number of base registers available on the CYBER, all addressing is absolute for global storage in CAMIL. As a consequence, if the size of preceding modules is changed by internal editing or redefinition of data within a preceding module, subsequent modules and any modules which refer to them will have to be recompiled also to obtain proper addressing. The declaration section must thus record the starting address assigned to each module because this affects the partial compilation decision.

As each declaration is processed, the TYPESPEC routine is called to build the structural description of the indicated type. If explicit TYPE identifiers are encountered, this routine merely references the existing definition. If compound types are structured, such as ARRAYs, RECORDS, or FILEs, a tree must be structured containing each of the imbedded types. A special routine, COMPSPEC, is provided for record and procedure headings since these are very complex in CAMIL. Simple types such as subranges, type identifiers, and user-defined classes are handled by a routine named PRIMTYPE, meaning primitive type, whereas most other compound types are handled directly by TYPESPEC. The call to TYPESPEC returns a pointer to the type definition, which will be merely the head of the tree structure for compound type definitions. A routine called COMPTYPE is available to determine whether two types are compatible and is used extensively during executable code compilation to determine whether the types of two operands are agreeable or whether the type of an expression encountered is the type anticipated. This routine is also used during declaration compilation to compare the types of constant expressions used for initialization with the types of identifiers they are being used to initialize.

A side function of the TYPESPEC routine is to determine the size in words and bits needed to represent the indicated type of entity. When a type definition requests that storage be packed, TYPESPEC will use knowledge of addressing rules to determine the most efficient way of packing data together to minimal size without sacrificing accessibility. TYPESPEC will return, in the resulting type information, the size of the total definition encountered. This information is used by the allocation routines in VARDEC, CONSTDEC, and NAMEDEC to determine storage allocation for the indicated defined identifiers. If an initialization expression is encountered, EXPRESSION is called with the TYPESPEC of the identifier to be initialized and told to attempt to accept a constant expression of the indicated type. If this attempt is successful, EXPRESSION will have computed the value of the expression at compile time and placed the resulting value at the address in the object code buffer correlated with variable being declared. If no initialization is found, the compiler will "zero" the associated size of storage in the object buffer. In this manner, values for all constants and initialized variables are generated as the declarations are compiled. If upon completion of all declarations, all initial values are zero (a very common situation), the compiler will note this fact in the module directory, rather than saving the initial values so the program loader may use this information to initialize the module data areas to zero. Because resulting initial values are built into the code buffer, data areas are currently limited to the size of the code buffer, but minor modifications could move this buffer to ECS, allowing it to expand considerably in size.

Declarations local to a procedure or segment are located at the beginning of the body and are compiled by calling DECLARATION for every module. The same process described above takes place with the exception that the PASCAL heap is marked prior to activating DECLARATION. Since any items defined locally are unknown outside of the body, any type data or symbol table entries created inside DECLARATION are not needed after it has been compiled. Thus space allocated for this purpose can be returned after the module has been compiled, reducing the total space requirement for compilation.

Statement Compiler

The executable statement section consists of a manager routine, STATEMENT, that identifies which type of statement is being compiled, and a set of procedures which each recognize and compile one type of CAMIL statement. Each of these routines recursively calls the EXPRESSION or STATEMENT routine to compile embedded expressions or statements. Each routine is responsible for consuming an entire statement of the type in which it specializes and recovering from any errors which are found in the statement it compiles. In order to prevent any statement routine it calls recursively from running away and consuming part of the statement handled by the calling routine, a set of stop tokens is passed recursively down through the calls. Each routine called adds its own stop symbols to the set it receives and passes the result to any routine it calls. No called routine may cross any token in this stop set while recovering from syntax errors unless the token could legitimately belong to the statement compiled by the called routine. In this manner, multiple errors which might be caused by "eating" important reserved words, such as "END", "DO", and ";" are significantly reduced. Special logic to treat commonly encountered errors has been easily added to

the statement recognizing routines since each may be individually tailored without altering the compiler as a whole.

Each statement in CAMIL may provide unique opportunities for local optimization of the machine code. For example, in the IF statement, after executing the selector expression, machine registers will contain the same information regardless of whether the THEN or ELSE statement is selected. The IF statement routine takes advantage of this fact by compiling code for both statements as though the variables used in the selector expression are available in registers. This requires the IF statement routine to save the register status after EXPRESSION is called to compile the selector and to assert this information as STATEMENT is called for both the THEN and ELSE statements. Thus unnecessary reloading of registers may be avoided for both imbedded statements. In a similar manner, all other routines which compile statements perform various degrees of optimization as possible to improve the size and execution speed of code. Since this optimization is accomplished as the code is being generated, no subsequent optimization pass is needed and information about the expressions need not be saved for long periods of time.

The instruction set for the CYBER computer does not provide a relocatable conditional jump statement. The effect of this shortcoming is that branch instructions generated to implement statements such as IF-THEN-ELSE, CASE-DO-ELSE, FOR-FROM-TO-BY-WHILE-UNTIL-REPEAT-DO, file-operation-ELSE, and JUDGE-ELSE must be generated with knowledge of the absolute address where the code will reside at run time, or a relocating loader must be used to modify the code prior to execution. The CAMIL compiler uses absolute addressing, thus eliminating the need for code modification by a loader, but creating the potential problem of mapping code into the proper location. This problem is solved by generating code as though all segments and all procedures execute in the same area of central memory. Since only one segment is ever executing at a time, this causes no problem with segments. However several procedures can be executing simultaneously, so a solution is reached by adding a swapping action each time a procedure is called or returns. When one procedure calls another, the called procedure is swapped in from ECS onto the code for the calling procedure. Similarly, when it returns, the calling procedure will be swapped back in. Since the CYBER can swap memory approximately 10 times faster than it can execute code, the resulting overhead is quite low and is often necessary to perform anyway since the program is constantly being swapped in to central memory for time-sharing purposes.

Expression Compiler

The expression section of the compiler is responsible for the computation of all constant expressions and for the generation of machine code for all computable expressions. The implementation of procedure calls in CAMIL further requires that the expression routine generate all procedure calls, sentence calls, function calls, and all user declared prefix, infix, and postfix operators.

CAMIL resolves all expressions or subexpressions involving constants at compile time. This means that any time the expression routine finds two constant operands and an operator, it will merely replace these with the result obtained from executing the operator on the operands. Since the compiler runs on the same machine as CAMIL, the result is identical to executing the code at run time. This means that complex expressions involving constants may be used to define other constants or to assign values to variables. Since CAMIL allows multivalued data types, such as arrays and records, it also provides multivalued constants to use as values for these data types. To reduce the character size of these expressions, a repeat operator is available to denote the repetition of a particular record field or array cell value. When these expressions are constants used for initialization, the resulting multiple words of memory are defined by the compiler and an assignment becomes merely a multiword copy rather than code to pack all of these fields, thus saving a large quantity of code space.

The expression routine also generates the machine code to create multivalued expressions such as are used as the values of records or arrays. The CAMIL declaration section generates identical structual definitions for procedure parameter lists and record field lists with the result that any type of procedure call, i.e., operator, function call, sentence, regular call, is effectively an operator acting on a single record of

the type of the parameter list. The manufacture of such items on the stack is performed by a routine called composed expression ("COMPEXPR"). This routine is the heart of all procedure call activity and is the most complex routine in the CAMIL compiler. Because a record etc. may contain OPTIONAL fields which may or may not be present, COMPEXPR must repeatedly try to match the types of expression it is encountering with the allowable types of expression which may appear in any field position. It is this facility which provides the flexibility which allows the highly complex "write" and "accept" sentences of CAMIL to be defined in CAMIL rather than being hand-coded into the compiler as is commonly done for I/O statements. Doing it this way also makes this power available to users for performing their own extensions of the language.

The composed expression routine also performs another very important function needed to support the sentence extensibility feature. When a procedure or record definition includes a variant definition such as:

```
CASE PRIMTYPE p
 BEGIN
  INT: "INTEGER i";
  NUM: "NUMBER n";
  CHR: "CHAR c";
 END;
```

where PRIMTYPE is a class containing INT,NUM, and CHR, then whenever COMPEXPR composes an expression such as:

```
(...., 37.56, ....)
```

in which the number 37.56 falls into correspondence with the variant field, not only will the value of 37.56 be assigned to the variant, but the value of the corresponding tag "NUM" will be assigned to field "p". When such a variant definition is used to define the parameters of a procedure call or sentence object, the resulting routine may be called with any of the allowable types such as INTEGER, NUMBER, or CHAR, and the procedure can identify what type of parameter was passed to it by examining the field "p". Using a definition like this, the object of a sentence such as "write" is defined as an array of records each containing one optional variant field of the general type included above. Thus users of "write" may call the routine with any of the allowable variants and the compiler tells the write routine the type of each of the arguments passed through the CASE variant selector variable "p". The routine can of course branch appropriately on this type to CAMIL code to convert and print each of the allowable types. Since the elements of this array are optional, the CAMIL program can also test to see how many of the array elements have actually been composed and thus only process the elements which have actually been passed. COMPEXPR supports this by setting a field in the record which can be tested with the CAMIL "=" operator to see if the corresponding optional field is NIL.

The EXPRESSION routine is highly dependent on three other routines, LOAD, STORE, and SELECTOR, for obtaining and returning the operands it computes. For compatibility with PASCAL for data analysis purposes, these routines were obtained by modifying the corresponding routines in the PASCAL compiler (reference 2) to be compatible with CAMIL absolute addressing requirements. In this manner, it is possible to write CAMIL and PASCAL record definitions which exactly match in addressing field for field. This makes it possible to write CAMIL programs for interactive execution which record data for analysis by batch PASCAL programs. This is exactly the method used by the CAMIL program editor when it creates program directories which are in turn used by the CAMIL compiler and print programs. EXPRESSION is actually composed of five levels of recursively activated procedures which each implement the operators which occur on five different precedence levels. SELECTOR is used to generate the code necessary to calculate array, record, or name references, while LOAD and STORE generate code to actually place the selected operand into a register or insert it into memory. Since CAMIL provides that existing operators may be extended to new user defined types while retaining their normal precedence, each level of operator must also check for the presence of user redefinitions of the operator before rejecting an

expression. These operators implement the numerous CAMIL built-in operators such as "line", "col", "min", "sec" which are used to produce the highly readable CAMIL sentences.

V. CAMIL EXECUTION SUPPORT SYSTEM

The CAMIL run time environment consists of a collection of programs and routines written in PASCAL, COMPASS, PPU COMPASS, and CAMIL. While executing, the system occupies three batch control points (including the compiler control point), three peripheral processors, and SCOPE operating system modifications. Each of the six basic programs (three batch programs and three peripheral programs) are separate processes, and communication between the processes is accomplished through ECS and centeral memory buffers. The basic components of the system (excluding the compiler) are:

1. The terminal driver program: "DRIVER"
2. The CAMIL execution program: "EXECUTER"
3. The CAMIL File Manager.
4. The peripheral routines:
   a. The terminal communications program: "INO"
   b. The CAMIL program timer: "TMM"
   c. The CAMIL disk interface program: "DAB"
5. SCOPE operating system modifications.

Terminal Driver

The basic function of the terminal driver program is to provide the capability of cummunicating with the terminals. The central memory driver program is needed to analyze the keypresses and perform the high level asynchronous interface between the terminals and the CAMIL programs. However, the central memory program is incapable of direct communication with the I/O channels connected to the terminals, so another process is required. The driver program communicates with a peripheral routine (INO) through central memory buffers so that all terminal communications are taken for granted in the central memory program. The peripheral routine, in turn, performs the actual data link between the central memory buffers and the terminals through the proper I/O channels.

The terminal driver, "DRIVER" occupies one of the batch control points and is written mainly in PASCAL with a few COMPASS packing routines. It is broken into the following sections:

1. Initialization section.
2. Key input section.
3. Communication section.
4. Framing section.
5. Job scheduler.
6. Batch file manager section.

Each section is basically a separate section, but some interaction does occur between the job scheduler and other sections. The sections are implemented as single procedure calls for each section, so the main block of the driver calls each of the different sections.

Initialization Section

The initialization section performs the initializations of the variables used by DRIVER and also initializes ECS which is shared with EXECUTOR. The initializations are accomplished by having the driver call a peripheral routine to initiate another job at the executer control point which shares the driver ECS area. The job then initializes ECS and also places all of the variable initializations into ECS and the driver just does one ECS read to initialize all of its variables. Once all of the required initializations are completed,

DRIVER again calls a peripheral routine to initiate EXECUTER at its proper control point and then waits until EXECUTER completes its own initializations, at which time the system is active.

Key Input Section

The key input section of DRIVER interrogates the incoming keys from the terminals. DRIVER will echo,buffer, or ignore the incoming keys depending on the state of the program for the corresponding terminal. The key section supports features in the accept sentence which:

1. Allow the user to limit the number of keys which may be accepted.
2. Process the response when a specified number of keys have been accepted.
3. Limit the keys to upper or lower case letters or to digits.
4. Prohibit keys from automatic echo.
5. Inhibit the automatic response input arrow.
6. Accept input from the touch panel (a rectangular ring of infrared light emitting diodes along the top and one side of the panel face with corresponding sensors on the opposite sides, which can detect a finger touching the screen at 256 discrete areas formed by the intersection of 16 vertical and 16 horizontal light beams).
7. Schedule input automatically upon each keypress to support representative echoing of keys pressed in a manner selected by the program author. The key section also intercepts active function keys and processes the synchronous or asynchronous meaning of these keys if they are currently defined.

Communication Section

The communication section of DRIVER receives messages from the CAMIL programs executing in EXECUTER. The typical messages sent to DRIVER indicate some type of action the job is waiting on: such as user input, a pause, a file operation, or just another time slice. DRIVER will decide what the job is waiting on and will perform actions requested by the job. A job which is requesting a new time slice will be sent to the scheduler, where it will be assigned a priority according to its utilization rate.

Framing Section

The framing section of DRIVER is a synchronous routine which must emit output for the terminal interface program INO every 1/60th of a second. Each terminal can receive at most one 20-bit parcel every 1/60th of a second, so the framer must break down the output going to the terminals into these 20-bit parcels. It must also keep track of what parcels have been sent and to which terminal each parcel is to go. When a terminal detects a parity error in a parcel it receives, it rejects the parcel and begins transmitting data to the central interface unit that it has done so. The framer recognizes this condition and requests that the terminal tell him the number of the last frame correctly received. DRIVER then resumes transmission with this parcel, thus insuring that no data are lost at the terminal.

Job Scheduler

The scheduling section of DRIVER contains three separate queues for scheduling CAMIL jobs. A job is placed in one of the queues, first depending on the reason the job is being scheduled (keypresses being a top priority) and secondly depending on the utilization rate of the job in processing milliseconds per real-time seconds. Jobs with low utilization rates ($\leq$ 5 ms/sec) are placed into the top priority queue, jobs whose rate is $\leq$ 10 ms/sec go into the next queue, and the rest of the jobs are placed into the final queue. If a job utilization rate is $>$ 15 ms/sec, it is placed into a wait queue for as long as it takes to lower the utilization rate to $\leq$ 15 ms/sec. This helps keep CAMIL program response times consistent with each execution and less dependent on the system load.

Jobs are removed from these queues by DRIVER and placed into an execution array which is monitored by EXECUTER as space is made available in the array through the execution of jobs already in

the array by EXECUTER. Highest priority jobs (the first queue) are given three slots in this array because of their low utilization rates. The next two slots are for jobs from the second queue (only two slots due to a higher utilization rate), and the last queue gets only one execution slot. A simulation of this queueing system (Reference 5) shows that the response times do not deteriorate significantly as system load increases because utilization rates are limited and priority is given to jobs using reduced CPU time.

Having the scheduler within the driver program allows the driver to schedule CAMIL programs when their accept or pause criteria have been met, and to allow a fast response to user key inputs by giving them a high priority. The imbedded scheduler also allows the driver program to initiate a new CAMIL job (known when a keypress arrives from a terminal not yet defined to the system) and to schedule the CAMIL batch file manager program (when I/O requests from a batch job are requested).

Batch File Manager Section

Requests for CAMIL file manager operations are placed into a central memory buffer in the driver by the SCOPE file manager modifications. There is a CAMIL job associated with each batch control point, which the driver schedules each time a file manager request is received from its associated control point. The CAMIL job then calls on the file manager to complete the batch job file manager request. When the request has been satisfied, the CAMIL program notifies the driver that it has completed the file operation, and the dirver then suspends the CAMIL job until another request is made. A simple modification to the SCOPE scheduler prevents the batch requesting program from further execution until the I/O request has been accomplished by CAMIL.

Executer

The "EXECUTER" program occupies the other control point of the CAMIL run time environment. EXECUTER operates in two modes: system and user. The system mode of EXECUTER performs the system initializations and swapping of the CAMIL programs. CAMIL programs execute in the user mode, after the system mode swaps in the job.

The EXECUTER is written mainly in PASCAL, with some routines and CAMIL primitives written in COMPASS. The CAMIL program area is also declared in COMPASS to guarantee that the CAMIL program area is always in the same absolute memory space even though the relative addresses of EXECUTER variables may change. EXECUTER occupies 55,000 octal words of central memory space, which includes all static memory requirements for 60 CAMIL jobs.

System Mode

The system mode of EXECUTER has its own memory space allocated for the run time stack of the system. System tables and variables which are stored in central memory are directly accessible to the system mode. Also contained in this area are the address tables of system information which is stored in ECS, such as program control blocks, system routines, and system shared variables. The system mode of EXECUTER is broken into three procedures: swapin procedure, swapout procedure, function key processing procedure, and one main program block.

The swapin and swapout procedures perform the swapping of CAMIL programs. Once a CAMIL job is scheduled, the swapping procedures are called to perform any necessary swapping to execute the CAMIL job.

Before control is passed to the CAMIL program, the function key procedure is called if the job is being scheduled due to a function key press. The function key procedure will search through the program run time stack to find the latest activation of the pressed function key. Once the activation of the function key is found, the function key processor will unwind the stack (if necessary) to the function key activation level, and set the return address to the function key definition address. Thus when control is passed to the CAMIL program, the function key processing code is executed.

The main program section of EXECUTER searches the execution array (which the DRIVER fills) for CAMIL jobs to execute. When EXECUTER finds a job to execute, the proper procedures are called to swapout the previous job (if necessary), swapin the new job (if necessary), and perform any function key processing (if necessary). Control is then passed to the CAMIL program, and the EXECUTER enters the user mode. When the CAMIL program re-enters the system mode, EXECUTER searches for more jobs to swapin.

After searching the execution array, EXECUTER will check the file operation pointers to see if any physical I/O operations have been completed. If there have been, EXECUTER will swapin the jobs that have had any I/O operations completed. When the EXECUTER has no more jobs to execute and no I/O operations have been completed, it relinquishes the processor to allow the compiler and batch jobs to have chances for the processor.

User Mode

The user mode of the EXECUTER uses the CAMIL program run time stack for variable storage. The CAMIL program is swapped into a section of EXECUTER central memory space and given control of the processor. When the CAMIL program is swapped into central memory, the timing routine is notified to begin timing the processor usage of the CAMIL program. The CAMIL program is then allowed full control of the processor and must voluntarily relinquish the processor back to the EXECUTER. If the CAMIL program does not release the processor before its time slice ends, the timing routine notifies the CAMIL program to release control of the processor by setting a flag which the CAMIL program automatically queries through code generated at points where the program might otherwise enter an endless loop.

The CAMIL language has many built-in primitives which need to be accessible to the user program. Most of the primitives could be coded in CAMIL itself, and many are, but for efficiency sake, there are also some coded in COMPASS and PASCAL.

The CAMIL primitives which are coded in COMPASS include the arithmetic functions (SIN,COS, etc.), file manager linkage, procedure calling linkage, string operators (concatenate, search, etc.), and conversion routines (string to integer, integer to string, etc.). Linkage is made to these COMPASS primitives through special handling in the compiler which places the parameters in special registers. The JUDGING primitives, reprieve logic, and control transfers (system→user mode) are also written in COMPASS. The total set of COMPASS primitives occupies 2,316 octal words.

Some of CAMIL primitives are programmed in the PASCAL language and are physically located within the EXECUTER support program. The linkage to these procedures is similar to normal CAMIL procedure linkage so that the compiler need only make a minor change in the normal procedure calling sequence to call a PASCAL primitive. The local variables of the PASCAL primitives are placed onto the user run time stack in the same manner that local variables of CAMIL procedures are added to the stack. Because PASCAL procedures do not have code compiled in to check the CAMIL time slice flag, those critical routines which may use resources common to all programs will not be interrupted until they have completed an entire logical process, although their execution time will be allocated to the CAMIL program calling them. The file manager, write sentence, accept sentence, and procedure and segment swapping are all implemented as PASCAL procedures.

Most of the CAMIL primitives are written in CAMIL itself. These procedures are stored in a section of ECS which is reserved for system procedures. When a system procedure is called, the procedure swapping mechanism sees that the called procedure is a system procedure and swaps it in from the system procedure area. The system procedures also have a special central memory area which they are swapped into. This is to allow the system procedures to reside in central memory longer and reduce swapping. Some of the primitives which are implemented as CAMIL procedures are write large, circle, draw, erase, slide, echo, ok, no, sized, pause, connect, dots, external, and all of the system functions available with the AUTHOR key (monitor, talk, autopsy, etc.). The write sentence has not been made a CAMIL procedure due to the many procedures used by the write sentence. Because the write sentence is used quite frequently, and many procedure swaps would be necessary for each call of the write sentence, it is resident in central memory as a PASCAL procedure.

Because the CAMIL code is machine code, mode errors become possible due to improper arithmetic operands. The reprieve logic of EXECUTER performs an interrogation of any mode errors. If EXECUTER was in the user mode (a CAMIL program was running) when the error occurred, the autopsy routine would be called to store data for an autopsy of the program. The CAMIL code also provides run time error checking of pointer values, array subscripts, and subrange values. The reprieve logic must also check for a mode error caused by run time arithmetic errors and properly report the cause of the error when it can be determined (the CYBER computer does not detect certain integer overflow errors). The compiler assists in the detection of logical errors by compiling code to check for the conditions mentioned above by compiling a jump conditional on the checked for condition. Rather than generating a jump to a specific error processing routine, the compiler creates an address field in the jump instruction to a nonexistent address, consisting of a high order address bit (to force nonexistence) followed by the line number in the program and the logical error number, all of which will fit into the 18-bit address field used in the CYBER computer. The resulting pseudo address causes the processor to halt and the CAMIL reprieve processor can then decode the "faulty" instruction into its actual meaning. Encoding the test in this manner saves more than 30-bits each time this type of test is performed and allows error messages to be related to the line in the CAMIL source program at which the error occurred.

File Manager

The CAMIL File Manager System is a completely closed file system (only accessible through the CAMIL system) and capable of handling many different file operations. The basic concepts of the file system are: to allow multiple access to files (any file can be accessed by more than one user); to provide a structured file concept (the compiler knows the formal definition of all the files in the system, so file use in programs must be consistent with the formal definition of the file); to provide indexing, direct access, and variable length files in an efficient manner; to allow batch programs to communicate with the CAMIL file system; and to provide a simple and uncompromisable file security system. All of the goals of the file system have been met, providing a powerful, efficient, and secure file system.

The basic logic of the file system is contained in one procedure (with nested inner procedures), and it resides in the CAMIL executer program. Other components of the file system are: the peripheral routine to communicate with the 844 disk controller, and CP monitor modifications and driver program linkage (to schedule the special CAMIL batch file manager interface program) to allow batch programs to communicate with the CAMIL data base.

The basic design of the CAMIL file system is such that it provides a powerful file concept in the most efficient manner possible. Some of the file constructs were limited from the original implementation in order to keep the file system efficient, but sufficient flexibility was insured to perform all of the desired operations. This type of implementation strategy led to a highly successful and easy to maintain file system.

All but an insignificant portion of the file manager logic is programmed in PASCAL and is resident in the EXECUTER program. The logic is broken into small procedures to perform each of the different file operations (READ, WRITE, DELETE, etc). These procedures in turn share other common procedures to perform operations such as record number verification, physical buffer allocation, and physical disk I/O. Each job which requests a file operation enters the re-entrant file manager code, and since the file manager code executes in user mode, all of the needed local file variables are placed onto the CAMIL run time stack. Because the PASCAL file manager code cannot be interrupted by another CAMIL job (the PASCAL code decides when to relinquish control), no synchronization is necessary between jobs requesting file operations.

Because the file system is shared, all current information about system files is kept in ECS. This allows all of the jobs requesting file operations access to the information without the need to reserve storage space in the run time stack of each job. The system file definitions for each file defined in the CAMIL system are stored in ECS, so that a file request can easily be verified without a disk request. Also while a file is open, all of the extra information which is needed for an open file (buffers, bit maps, etc) is contained in ECS and referenced through the resident file information.

There are three types of files: direct access, indexed, and variable length files (though direct access and indexed files can be accessed sequentially). The most common type of files are the direct access files. Direct access files provide the capability of accessing fixed length records at very high speeds. This is accomplished by being able to compute the physical disk address from any given file address, so that the only physical I/O required (sometimes none is if the record happens to be in a buffer) is the actual data transfer (note: The record bit maps must also be backed up to disk when writing a new record, but the backup operation is part of a single I/O request). Because direct access files allow packing of records (more than one record per physical block), two physical operations could occur for a write file operation on a packed direct access file (one to read the physical block, insert the new data, and then write the physical block back out).

The indexed files are designed to provide a high speed indexing method to structured files. They are fixed length records (preferably small records), and the entire file resides in ECS. Therefore no physical requests are necessary for read operations, and only one request (to back up the file on disk) is required for write operations. The typical use of an indexing file is for indexing purposes. The record associated with the desired index may contain access flags, status sets, and direct access file addresses. The direct access file addresses are used to associate data located in direct access files with the specific index. The direct access address can then be used during the processing of data associated with the current index so that all further file operations are as efficient as possible using computable disk addresses. This approach eliminates the need for index searches and index blocks (which consume time, space, and disk accesses) without imposing any real burden on the programmer.

Variable length files provide a means of storing records of variable lengths. They are similar to direct access files in that disk addresses are directly computed from the addresses of the records. The main difference is that one cannot direct where a record is to be stored when writing out a record; instead the file manager assigns a new record number each time a record is written; Also, it will delete the old record (if rewriting a record). This is necessary because it may not be possible to fit a record back into the same record position it came from (the record could become larger), so the file manager will automatically delete the old record and insert a new one, returning the new record's address. The number of necessary physical I/O requests per record access is at most one per request (none if the record is already in a buffer), since all disk addresses are computable and the disk driver routine will read in only the needed number of sectors for variable record reads. As with the direct access write operations, backup of record bit maps is also part of a write request; thus, only one pause for physical I/O is necessary per operation, although more than one transfer may take place.

The file manager has its own peripheral routine to handle all of the CAMIL data base requests; therefore, the disk addresses computed by the file manager are directly handled by this routine. It is the use of this special routine which also allows the record bit maps to be stored in the same request as a write request, thus cutting down on swapping and waiting time overhead of producing two physical requests. The data path between the peripheral routine and the file manager is also minimized since the peripheral routine transfers the data directly to or from the file ECS buffer.

Requests from a batch job requesting a file manager operation are processed identically to CAMIL file manager requests except for the data transfer portion. When data are to be transferred to or from the batch program, the CP monitor modifications are called to perform the transfer. In CP monitor, the data are simply transferred directly to or from the file ECS buffer from or to the batch program central memory buffer. Thus the data are transferred in a most efficient manner between file ECS buffer and batch central memory buffer without any need of transfer buffers or extra movement of data.

For each file manager request that a program makes, the file manager checks to see if the program has permission to perform the requested operation. If the program does not have the proper authorization, a file security error is generated and the operation does not occur. File security is accomplished by associating a program name with a set of permissible file operations. Each program which is to have its own set of access privileges to a file must be placed in the file security list by the FILEEDIT program. A default

set of permissible file operations can also be specified, in which case any program without special privileges to a file would assume. In this way a file can have a nondestructive set of default privileges so that other programs can be allowed to inspect the file without giving specific read permission to each individual program. Because the file manager operations are defined in the PASCAL compiler as well as in the CAMIL compiler, the file security by program name also holds for batch file manager requests. Because of this, and the fact that only CAMIL and PASCAL programs can access the data base, the security of the CAMIL data base cannot be compromised by any method, since only specified programs can be authorized to access data base files, and there are no passwords which can be stolen.

Operating System Interface

The most extensive modifications to the SCOPE central memory monitor program have been made to allow batch jobs to communicate with the CAMIL file manager system. These modifications are incorporated into the RA+1 section of the CP monitor because of the expected frequency of use of the file manager requests.

A batch job issues a request to the CAMIL file manager by calling DIO (resident in RA+1) which passes pertinent file information to a batch file buffer in the driver. (The batch job is suspended until the file manager completes the request, at which time the job is resumed.) The driver then schedules a CAMIL job which calls the file manager routine to perform the relevant file operations.

The file manager handles batch and CAMIL file manager operations in a similar manner, except when transferring the actual data to or from the program's buffer. In the CAMIL case, the file manager can simply read or write from the file ECS buffer into the program central memory buffer. The batch case however requires a call to "ITO" (RA+1 resident) to perform the transfer between ECS and central memory (the central memory space belongs to the batch job). In both the batch and CAMIL case, however, the data are transferred between ECS and central memory only once.

The CAMIL system also requires special scheduling of the driver, EXECUTER, and the compiler. The driver must always have the top priority of any job in the computer because of its synchronous nature. The EXECUTER is next on the list of special priorities, since an interactive job requires a faster response than a batch job. The compiler must also be given a priority over batch jobs, since an interactive user is waiting for the results of the compilation. Modifications to the SCOPE scheduler were made to accomplish the special scheduling requirements with minimum interference with the normal scheduling of batch jobs.

Peripheral Processor Routines

INO

To communicate with the terminals, two channels are dedicated to the system terminal hardware interface units. The "INO" PPU routine communicates between the driver and terminal hardware interface units through central memory and the data channels, respectively.

One channel is dedicated as an input channel. The INO routine queries the channel for incoming keys. When a key is received from a terminal, the hardware will place the key (along with the terminal number the key came from) on the channel. INO will then place the incoming information (assuming no parity errors occur) into a circular central memory key buffer in the driver. The driver properly responds to the key strokes, either echoing or buffering, etc., depending on the state of the program running at that particular terminal.

The output channel operates in a synchronous mode, since the terminal hardware requires output for the terminals every 1/60th of a second. The output channel can send each terminal only one 20-bit parcel each 1/60th of a second. INO awakens the driver to prepare a stream of these parcels, encoded with terminal number and data, to meet the terminal hardware demands. Even if no data are to be sent to a terminal, the hardware demands at least one parcel to be sent to an undefined terminal every 1/60th of a second.

Once the driver has created a stream of parcels to be sent to the terminals, INO reads the information from central memory and then transfers the information over the output channel to the terminal hardware interface unit. The interface unit breaks down the information and sends the data to the proper terminals.

DAB

The CAMIL data base is totally separated from the SCOPE file system. This separation was accomplished by developing a new I/O routine which processes all CAMIL data base requests. This routine communicates with the CAMIL executer through a request buffer which is prepared and monitored by executer. The new routine transfers data from the CAMIL data base on disk directly into a data buffer in ECS where it is retrieved by the requesting program as soon as it can be rescheduled. This eliminated much of the overhead and unneeded data shuffling incurred with the CDC supplied software. It also provided greater isolation between the two systems (CAMIL and SCOPE). The drives used for the CAMIL data base are not known to the SCOPE system, and the two systems are thus mutually inaccessible, except through programs capable of attaching to both data bases.

Data base I/O requests are handled on a first in, first out basis. File manager (FM) determines when a physical I/O request will be needed to satisfy a CAMIL request for data. File manager constructs this request and places it into the DAB request buffer. Essential items in the request are the logical pack number, cylinder number, initial sector number, the source/destination ECS address, and the number of sectors requested (for fixed length records).

The CAMIL data base consists of up to eight 844 disk packs. Each pack has a logical pack number (0 to 7) and a pack name. Each pack is considered by FM to be error free. FM sees a pack as 410(0 to 409) cylinders of usable space. Each cylinder is a logical set of 452(0 to 451) sectors. A physical cylinder has 456 sectors, the last four of which are used by DAB to replace up to four defective sectors per cylinder, thus maintaining the illusion to FM that every pack is flawless.

The sector substitution table is initialized by a pack initializer PPU routine, IPK. IPK writes and subsequently reads each allocatable sector on the pack and manufactures a substitution entry for every sector which is incapable of being reread. IPK also blank labels the pack, so that it can be permanetly labeled by the FILEEDIT program which is used to define the content and structure of the CAMIL data base.

TMM

The peripheral routine which times the CAMIL jobs is TMM. The EXECUTER tells TMM when to begin timing its use of the processor and the time slice to be allowed. TMM will time the use of the processor, continually placing the number of time units used by the job into central memory. In this way, when the CAMIL job is swapped out, the processor usage is immediately available to the swapping routine, and no special call is required to get it. If the job uses more CP time than the time slice allowed, a flag in central memory is set, which all CAMIL programs periodically check, and the job will voluntarily relinquish control of the processor.

TMM also updates the current date and time in the CAMIL date and time areas when it is not timing a CAMIL job. This allows CAMIL programs to directly access date and time information through system defined variables instead of special procedure calls usually found in other programming languages.

VI. CAMIL AUTHORING SUPPORT FEATURES AND AIDS

Because CAMIL is a highly flexible language, it was desirable to implement some system functions in CAMIL itself. All system level operations controlling access to the CAMIL system are performed by CAMIL programs. User LOGON passwords, system file definitions (including security access privileges), program editing, and even CAMIL program loading are all performed by CAMIL programs. Because these programs

are written in CAMIL, they provide an "intelligent" interface between the user and the CAMIL system and can be easily updated to reflect system changes. CAMIL programs provide the user with menus, help when requested, and interrogation of illegal requests.

One important facility is not a separate program but is imbedded in the EXECUTER program. This facility allows the author of a program to interrupt execution of the program by pressing an "AUTHOR" function key on the keyboard. The author key allows him to immediately autopsy the program, look into the data stack of the program, restart the program, communicate with other terminals, or monitor the activities of another terminal in the system. The monitor function provides for future access to a number of interactive breakpointing and analysis facilities which may be added to the system.

The function of each of the major programs used to implement the system will now be explained.

LOGON Program

When the system is running under CAMIL, every terminal is established as either unused or running a CAMIL program. When a terminal is powered up, it emits data to the computer indicating this condition. The CAMIL system establishes a data area for the terminal and begins executing a program called LOGON. In anything further done at that terminal, it will merely be jumping from one CAMIL program to another, i.e., LOGON→LOADER→EDITOR→LOADER→USERPROG→LOADER, etc. The LOGON program initializes the terminal and identifies the user by associating him with his user privileges through his LOGON ID and security password and information. His status in the system and everything he is permitted to accomplish are controlled by this information. As additional security is needed, it is provided by the concerned programs, which protect the data base, and apply restrictions based on security data in his user records. As an example, certain functions might only be allowed to be performed by certain programs run by certain people at certain terminals in certain buildings during certain times of the day. In this manner, multiple restrictions are placed on critical data areas so that penetration of a single person's personal data is inadequate to compromise system security. Final control is retained by restricting data base access to programs by name (each program name is unique) so that if a program could be copied and modified to remove some security checks, it would still be denied data access by virtue of being a different program.

The role of the LOGON program, in this process is to identify the person trying to log on, determine whether he is permitted access from the log-on site, and apply restrictions as recorded in his user records. Since the user will always be running some CAMIL program or submitting a batch program from some CAMIL program through the program editor, security is retained by the CAMIL system.

The LOGON program also has such peripheral functions as to display run time error information if a CAMIL program must be suspended, display resource utilization factors and display lists of programs permitted to the user. The successful operation of the LOGON program depends upon a user data base generated by another program called the user editor which establishes user permissions.

Program Editor

All CAMIL programs are created and reside in the AIS computer. Programs are intended to be authored on-line and updated interactively. For this reason, a powerful but easy to use editor is an essential part of the programming system. The CAMIL editor was inspired by the PLATO IV edit program and was initially written in the original CAMIL language implementation. It has now been rewritten in CAMIL II, resulting in an approximately 50-percent reduction in source program size, although the original editor is retained for use when the old system is executing.

The editor is intended to allow modular program construction for ease of access without causing annoying specific actions to be performed to link the resulting program modules. To support this, the CAMIL system, local PASCAL compiler, and a print program have been written to use or disguise this modularity as appropriate, thus allowing the user to create modules corresponding to CAMIL or PASCAL routines or blocks of text. The editor has four primary levels of operation: program, directory, module, and textual.

Program level operations are those such as creating, copying, deleting, compiling, cataloging, printing, or checking the status of a program. These are accomplished on an entry page as options available through single keypresses. The most frequently performed step from this page is to enter the directory level of operation. All programs are divided into major directory areas; in the case of CAMIL programs, these areas are correlated with specific divisions of the program and given the names: Shared, Private, Procedures, Segments, Errors, and Autopsies. Each of these is merely an entry point to a chain of directory pages, any one of which can contain up to 30 entries and is linked to the subsequent and preceding directory pages. The directory is presented to the user as a menu of module names, each with a number that can be used to enter the module for editing. In addition, directory level functions, such as adding, deleting, rearranging, renaming, and copying entire modules, are performed at this level. Also module level print flags can be set for each module so that selective printouts can be accomplished by the print program. New directories may be added following or preceding the current page at this level. The user will normally select a module for editing by entering the module number on this page, which moves him to the module level of editing.

At the module level, the user is automatically provided a displayed set of lines representing the current location in the module. The user can set the number of lines that is seen by default to any number of lines that will fit on the screen; the system will initially display five lines. As the user moves forward or backward through the module, the lines that are displayed are numbered with small numbers from 1 to 31, and the user refers to lines by these numbers. Since the numbers are completely relative, lines may be added or removed, and the system will constantly display the updated text with familiar numbers that always appear on the same lines of the screen. Since these numbers are kept as small as possible, typing is kept to a minimum. If the user wants to see more lines than are displayed at any moment, this can be done by pressing the space bar, and the editor will double the number of lines currently on the screen and add this many more lines to the display. Lines already on the screen do not scroll or move as in some terminals so they can be easily read as new lines are being added to the screen. The user can move forward through the text by simply pressing the "NEXT" key, which will move the current location to the line following the line currently displayed at the bottom of the screen, and then redraw the screen to display the default number of lines.

Commands available at the module level allow entry to a textual level of editing in which lines may be inserted or replaced. In each of these modes, the user denotes insertion to begin after or replacement to begin with some line which is on the screen. The screen is redrawn with the referenced line near the top of the screen, and with the user cursor under the line of entry. In insert mode, the line inserted after is placed into a special copy buffer. Editing keys on the CAMIL keyboard allow this line to be copied wholly, word by word, or letter by letter into the user input buffer, along with any new characters to be added to the line. Other keys allow things copied or entered to be erased wholly, word by word, or a letter at a time. Still other keys allow the line, words or letters to be removed right to left from the input buffer as though they were being erased, but then returned to the screen and to the user input buffer at the press of another key. The combination of these keys allows existing lines in the module to be copied quickly to the point of a mistake from either the left or right direction, a correction to be inserted into the line, and the rest of the line to be copied without error. In replace mode the copy buffer is merely loaded with the line to be replaced so rapid updating of errors, without introducing new typing errors caused by reentering characters which are already correctly entered into the line, is possible. In either mode the user can skip over lines he does not want to change thus allowing him to easily move through an area containing errors and update or insert after each line as needed without having to redesignate with numbers which line he intends to alter next. Because these keys allow the user to directly edit the characters in lines, these keys perform the function of numerous string oriented editing commands found in more conventional editors. As a result, the only string oriented command is one with which a module may be searched for occurrences of a particular string, with optional replacement by another string by pressing a function key.

The commands available at the module level allow the user to move forward or backward by the number of lines displayed, to the beginning or end of a module, to the lines following the lines currently on

the screen, or to the following or preceding module by the press of a single key. Lines may be deleted by entering the starting and ending line number, or may be saved into a "save buffer" and carried to some other place in the module or into a different module, program or editor. In addition, groups of lines may be moved left or right a designated number of spaces to align them with other lines in the text; this is very useful in the editing of structured programs where indenting is often used to display program structure. In all commands referring to more than one line, the designated lines are encircled by the editor to confirm that the proper lines have been denoted before the operation is completed, thus giving the user a chance to change his mind before making a major error.

To simplify program storage, changes made to a module are not recorded on disk until the user leaves the module, at which time they are automatically recorded by the editor without any explicit action being performed by the user. A special escape is provided which allows the user to leave the module without storing the changes that have been made. This is normally used only when some major blunder has been made, such as deleting a large block of material by accident, which the user does not want to become a permanent change. If a module is emptied, it is automatically removed from the program directory, and if a new module is being created, it will be automatically entered into the program directory at the place it is designated to be added.

Another useful function supported by the editor is the automatic tab function. In automatic mode, the tab key will indent to the line which is being inserted after or replaced; this is useful for indenting structured programs or for entering indented textual material. A manual mode is also available where fixed columns specified by the user can be used when tab is pressed; this is useful for editing programs written in assembly language or for entering column sensitive data.

To assist in the development of structured programs, the editor searches for leading BEGIN and END symbols and the special CAMIL begin-end characters. When vertically paired symbols are found, the editor will automatically connect them with vertical lines each time the screen is redrawn. (This may be seen in the CAMIL examples included in the appendix.) An automatic grid function is also available which will draw vertical lines at designatable character intervals each time the screen is redrawn, to assist in placing column critical data at predetermined positions when the terminal is used as a pseudo keypunch machine for card format oriented data entry.

Automatic Error Mode

When a CAMIL or PASCAL program is submitted for compilation, the editor generates a request for the compilation by routing the request into the system input queue. The request includes only the name of the program to be compiled and any unusual parameters which are to be applied. The corresponding compiler obtains the program source by reading the program level directory, then the module directory pages, and in turn the source modules from the data base. No physical medium other than disk storage is used to retain the source information. To be consistent with this philosophy, the compilers do not generate program listings as the programs are compiled, but rather, if an error is encountered during compilation, an entry is made in an error record, indicating the module number, line number, column number, and error number of the encountered problem. At the end of the compilation this error record is recorded on disk for use by the program editor. When the user requests to see the error module directory, the error module is used to read up the module containing the first error, set the current line position to the line containing the error, draw a pointer to the place in the line where the error was discovered, and display the error number and an English description of the meaning of the error message at the bottom of the screen. (A hard copy of this display is included in the appendix.) In this manner, the author need not be at the central site with the printer to use the system. The resulting environment is much faster to use than a paper or screen equivalent of a listing with error messages embedded in the program text. This is particularly true as programs grow in size. (The AIS adaptive model used for student lesson management takes approximately 45 minutes to list). Function keys allow the user to request the display of the next error as needed and then to go to another module to fix the problem, such as an undeclared identifier, without causing problems. This facility combined with rapid partial compilation can reduce complete turnaround cycles to less than a minute.

The editor also provides access to autopsy reports generated when a program fails in operation. As explained in another section, the entire data context of a program is saved in the event of such a failure. The autopsy program mnemonically dumps these data, and a source module is constructed for each local and global data area and for the built-in system variables for the program. It also builds a directory for these source modules so that the user can select which data area to observe in the same way that a module is selected to edit when editing normal program sources. These directories are also strung together so that the user can look at all of the autopsies which have occurred, independently of where the program may have been running within the AIS network. In this manner, field problems are returned to the program author, who then has a description of what was happening at the moment of failure, even though the author was not physically present at the time. All normal editing functions are available so the author may search for desired identifiers or values or may scroll through the autopsy looking for something which seems abnormal. The combination of these two interactive debugging aids greatly enhances the usability of the system, particularly for remote program development.

The editor program, which was written in original CAMIL, was translated into CAMIL in about two work weeks, and reduced in size about 50%. The resulting program is approximately 8,000 words of source code (2600 lines), compared with 32,000 words of source code (5400 lines) for the program written in original CAMIL. The resulting decrease in line size is due primarily to the more efficient syntax and sentences of CAMIL II, and the additional reduction in code size is due to a 35% improvement in source code storage density in CAMIL II editor format.

The User Editor

The file of information used by the LOGON program is created by another program called the user editor. This program allows an authorized person to create and modify records for other people. Naturally, administrative controls must be applied, controlling who has the ability to extend this privilege to other persons, but this is enforced by the user editor which is the sole program that can edit the user file.

The user editor will not be explained in depth, but it contains the necessary displays to establish, survey, delete, and modify user records.

File Editor

The CAMIL file system is managed interactively through the program FILEEDIT. With this program, file definitions are interactively created, edited, and deleted. The resultant file definition file is used at system initialization time to load system file information into ECS.

When creating a new file definition, the file editor solicits information (file type, record size, buffer size, number of buffers, security privileges, number of records, etc.) required to define a file. From the obtained information, the file edit program computes the total disk storage space required to hold the file, which is then used by the program when allocating physical disk space for the file.

When the user is satisfied with the file definition, physical disk space for the file must be allocated. The user may optionally direct where it will be physically located (by disk pack and cylinders) or may allow the FILEEDIT program to find the required disk space.

The file edit program also provides for general disk maintenance and disk allocation updates. Disk packs which have been initialized by the IPK routine can be labeled by the file edit program, making them ready for use in the CAMIL file system. Also allocation maps for each disk pack can be inspected and changed by the FILEEDIT program. This allows the status of each disk pack to be examined prior to allocation of a new file.

Autopsy Program

A CAMIL program in execution presents a pattern of information on a terminal screen which the author can observe to partially determine whether his program is executing correctly. Simultaneously, variables internal to the program, but not visible to the author, are undergoing continuous change. It is often very desirable for the author to observe this internal state, but this is quite difficult to accomplish, since normally the program would have to be temporarily modified to display these data, along with the desired screen output of the program. It would be highly desirable to have a tool which would display this information at the request of the user, without requiring modification of the program. It would also be very timely to apply this tool in the event of an unanticipated failure of the program during execution.

The dump is such a tool but has until recently been as crude in form as the programming languages it has served. The post mortem dump implemented by Sandmayr (Reference 6) has provided dump-like information in a mnemonic form for the simple data types supported by PASCAL. The CAMIL autopsy report extends the basic notions of the PASCAL PMD to include all user structured data types, such as packed records arrays, files, sets, and classes. The CAMIL autopsy can also be taken any time during normal execution of a CAMIL program by pressing the AUTHOR key and requesting an autopsy.

When an autopsy is requested, the state of the program in central memory is written onto the data base for presentation to the autopsy dumper. The dumper will use compiler generated descriptions of the address space of the program to produce a mnemonic dump of the data area of all routines active at the time of the autopsy. It also generates the calling sequence of active routines and attaches all of this information to the program directory. The author can use the program editor to examine this information at will. The default autopsy covers all variables in the program, but compiler directives allow the autopsy to be selectively omitted for items in which the programmer has no interest.

Print Program

Because the character set for the CAMIL system includes 124 hard printable characters, a special printer chain is needed to print all of the character graphics used by the system. This special chain relinquishes some redundancy of frequently used characters in order to make positions available for the nonstandard graphics (print slugs) used for CAMIL. The absence of these slugs causes the printer to run more slowly, especially when CAMIL programs, including characters which appear only once on the chain, are listed. To counteract this factor, a print program was written which is capable of reading CAMIL directories and source modules, and printing the full character set on the printer in a unique two-page format.

Lines in CAMIL modules are never more than 60 characters in length since the AIS terminal screen allows only 64 characters total, and four of these are used by the editor at the left margin for line numbers and spacing. The line printer is capable of printing 136 character lines across a 15-inch-wide continuous paper form. To make the most of this combination, the print program prints two images side by side on each sheet of line printer paper. Because the print time for each line is determined primarily by the time waiting for all needed slugs to pass over positions where they are to be printed, printing a wider line has little effect on the printer speed compared to the need to wait for the full printer chain to cycle by each line. The resulting printout is thus twice as wide and half as long as the normal format and has the further unique property that it can be burst and each page folded upon itself, producing a book-like format which is much more convenient for program documentation. The major operational benefit of this format is that the printer runs almost twice as fast on these normally slow listings and uses half as much paper.

The print program also prints a program summary at the end of the listing which cross-references modules to the page of the listing where the module was printed. Pages are automatically numbered at the bottom and module line numbers and headings can be printed or deleted at the request of the user. The print program is written in PASCAL and attaches to the CAMIL data base through the batch program interface described in the file manager section.

VII. CONCLUSIONS

The language described in this paper is a workable usably implemented language. It reflects qualitative improvements in CAMIL derived from experience with the current operational implementation of the language. These improvements were sufficient to allow a more than 50% reduction in the size of the program editor which has been translated into the new format as a test case program. In addition, the resulting program appears to run both interactively faster (subjective observation) and consume less computer time during execution. The program is also significantly more readable due to the extensive use of the CAMIL user sentences and improved file structures. We feel that this saving is typical of savings which could be realized if the current system was converted to the new language format and that the greatly improved compiler performance would facilitate such an effort and future applications of AIS to new instructional areas.

A pivotal question which arises when such an effort of this type is contemplated is whether the benefits of such a conversion outweigh the costs in time, effort, and interference with the operational environment. If the AIS load should increase, major improvements would be needed to handle the additional load imposed upon the computer, demanding either additional hardware or improvements in software. If such an increase was to occur, an alternative to an increase in hardware performance now exists, along with qualitative improvements in development facilities.

In the event that demand for AIS computer services does not expand, or if it assumes a different direction away from the central, research oriented form that is currently implemented, we have nevertheless gained significant knowledge of the implementation approaches to use in future developments and of the types of interactive aids which should be included in future systems.

REFERENCES

1. Wirth, N. The programming language PASCAL. *Acta Informatica*, 1971, 1, 35–63.
2. Ammann, U. PASCAL-6000 compiler.
3. Sherwood, B. *The TUTOR language*. Computer Based Education Research Laboratory, University of Illinois, Urbana, Illinois.
4. Stifle, J. *The PLATO IV architecture*. CERL Report X-20. Computer Based Education Research Laboratory, University of Illinois, Urbana, Illinois, April 1972.
5. Krivacic, R. Refinement and implementation of simulation system. Masters Thesis, University of Colorado, Boulder, Colorado, April 1978.
6. Sandmayr, H. PASCAL post mortem dump program.

APPENDIX A: PROGRAM EXCERPTS

Several program excerpts are included in this appendix to display something of the CAMIL environment to the reader. Unfortunately, the extreme responsiveness cannot be captured on paper nor by a sequence of frames showing progress through a program.

1. Display of a syntax error as produced by CAMIL editor automatic error display mode. The editor user presses a single key which causes the editor to read up the module containing the next error and show him an English description of the error, which he can then correct.

```
Segments: picture          Space: 888      »
1     draw from 100,400 to 400,300;
2     draw to 200,100;
3     PLANE(270,400,300);
4     on line 31,col 5 write large 'Enroute display' sized 3.0,2.
      0;
5     on line 1, col 40 write

104:  identifier not declared
```

2. A simple typing drill program which places randomly selected words on the screen which the typist must correctly copy.

```
Segments: TYPO          Space: 747
1   CONSTANT  INTEGER lines←5, words←6, vocab←44;
2    ARRAY[Ø:vocab]OF STRING[8] wd←('puppy','enrich','done',
3     'angina','murky','clam','doodle','slap','carnage',
4     'cluster','granola','erase','zulu','quirk','aqueous',
5     'mauve','muddle','jiggle','epoxy','paraffin','gross',
6     'slick','slurp','amoeba','abilone','walrus','zero',
7     'sucrose','cranium','zap','zilch','xerox','lithium',
8     'serum','proton','silicon','tribble','clone','cut',
9     'in','do','timer','timid','pervade','mediate');
10  VARIABLE  INTEGER i,j,k,errs,chars,start; STRING[8] w;
11   PACKED ARRAY[lines, Ø:words-1]OF Ø:vocab  wdinx; CHAR char;
12
13  [IF [BACK] DO; start←SYS,CPUTIME; errs←Ø; chars←Ø; erase;
14  FOR i TO lines DO
15   FOR j FROM Ø REPEAT words DO
16    [k←RANDOM×vocab; wdinx[i,j]←k;
17    [write wd[k] on line(3×i-2),col (j×1Ø+2);
18  FOR i TO lines DO
19   FOR j FROM Ø REPEAT words DO
20    [w←wd[wdinx[i,j]]; chars←chars+LENGTH(w);
21    FOR k UNTIL k=LENGTH(w) DO
22     [accept rep with [noarrow,nocaps]; char←w[k];
23     IF J,KEY=char THEN
24      write char on line(i×3-1), col(j×1Ø+1+k)
25     ELSE
26      [unwrite w on line(3×i-1),col(j×1Ø+2);errs←errs+1;k←Ø
27    [UNTIL J,KEY=' ' DO accept rep with[noarrow]
28
29  on line 31, col 5 until [NEXT] write
30   "You typed ", chars:3, " chars with ", errs:3, " errors
31   CPUTIME=", (CPUTIME-start)/4.Ø96/chars:6:2, "Sec/keypress"
```

3. Original CAMIL code for a simple math drill program which randomly generates math problems and checks any wrong answer against the possibility of having performed the wrong operation on the displayed operands.

```
' Procedure: DRILL    Space: 747
  1   DECLARE INTEGER I,r,ans,opselect,oks,try;
  2   DEFINE STRING(1) ARRAY(4) op=('+','-','x','/');
  3
  4   Erase Screen; At Col 10, Line 5 Write
  5     'Welcome to MATH DRILL, Press NEXT to start'; Pause;
  6   REPEAT 10 TIMES DO
  7    BEGIN ON HELP DO (Write ans; Pause); try=3;
  8     Erase Screen; At Col 20, Line 30 Write 'Help Available';
  9     l=RAND()*12; r=RAND()*12; opselect=RAND()*3.+1;
 10     CASE opselect OF
 11       (1| ans=l+r; 2| ans=l-r; 3| ans=l*r; 4| ans=l/r);
 12     JUDGE
 13      BEGIN Erase Line 10;
 14       At Col 5, Line 10 Write l With Magnitude 2;
 15       Write op[opselect]; Write r With Magnitude 2; Write '=';
 16       Accept At Col 14,Line 10; try=try-1;
 17      END WITH
 18      BEGIN
 19       ans|(Write ' ok' For 1 Seconds;IF try=1 THEN oks=oks+1);
 20       l+r|(Write ' no did you add?' For 1 Seconds;
 21            J,FLAG=FALSE);
 22       l-r|(Write ' no did you subtract' For 1 Seconds;
 23            J,FLAG=FALSE);
 24       l*r|(Write ' no did you multiply' For 1 Seconds;
 25            J,FLAG=FALSE);
 26       l/r|(Write ' no did you divide' For 1 Seconds;
 27            J,FLAG=FALSE)
 28      END
 29     ELSE
 30      BEGIN
 31       IF try>0 THEN (Write   Answer was'; J,FLAG=TRUE;
 1         Write ans With Magnitude 4; Pause For 1 Seconds)
 2        ELSE Write ' no Try again' For 2 Seconds
 3      END;
 4    END;
 5  Erase Screen; At Col 5, Line 20 Write 'Number correct = ';
 6  Write oks; At Col 5, Line 21 Write 'Number missed =';
 7  Write 10-oks; Pause;
```

4. CAMIL II code for the same math drill program.

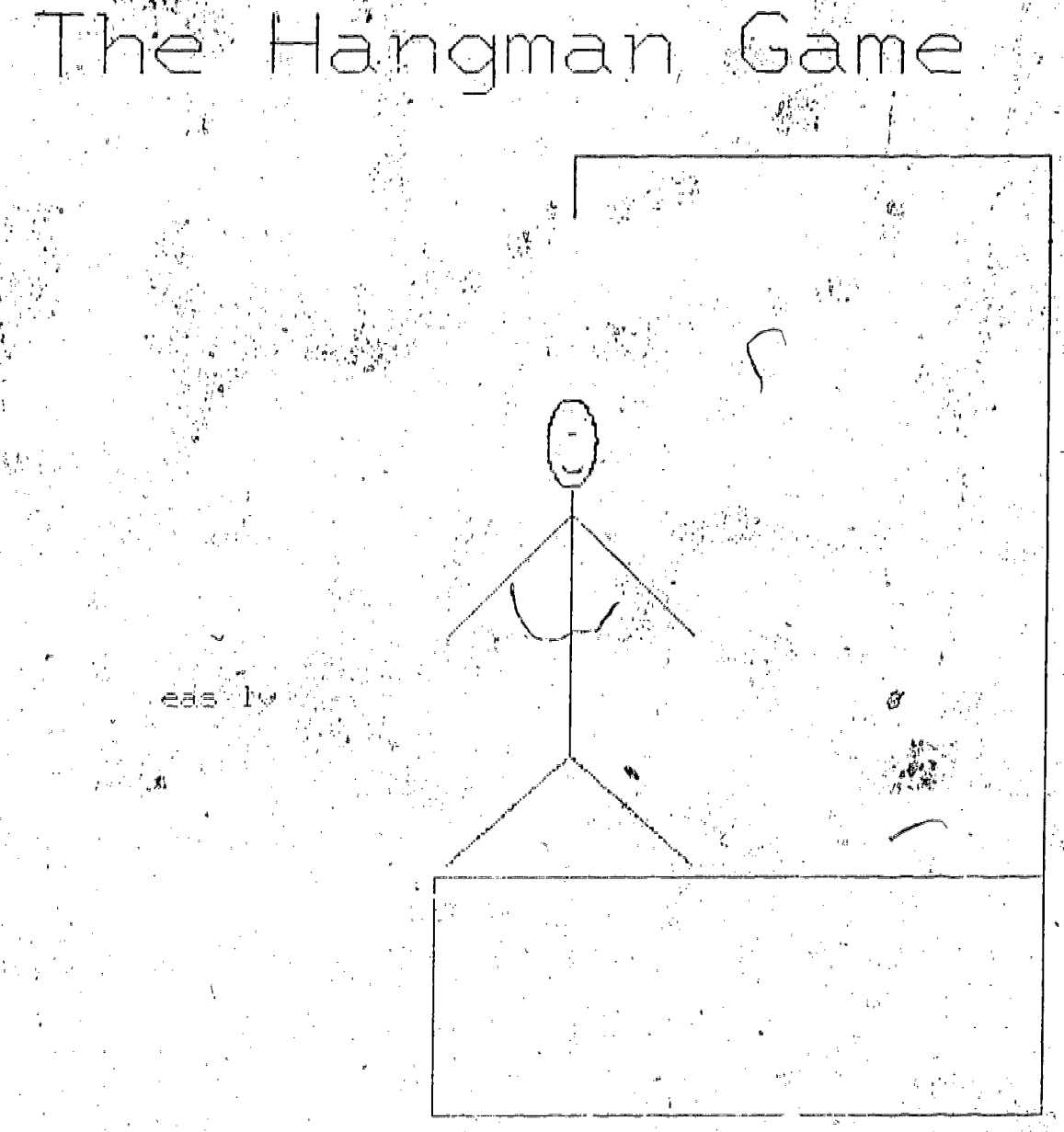
```
Procedures: MATHDRILL2   Space: 812   >
1   CONSTANT ARRAY[4] OF CHAR op←('+','-','x','÷');
2   VARIABLE INTEGER l,r,ans,opselect,oks;
3
4   [erase; on line 5,col 10 until [NEXT] write
5    "Welcome to MATH DRILL, Press NEXT to start";
6    REPEAT 10 DO
7     [IF [HELP,BLUE_BACK] DO write ans until [NEXT];
8     erase; l←RANDOM×12; r←RANDOM×12; opselect←RANDOM×3.+1;
9     CASE opselect
10     [1| ans←l+r; 2| ans←l-r; 3| ans←l×r; 4| ans←l÷r];
11    write "-HELP- available" on line 30, col 20;
12    on line 10,col 5 write l:2, op[opselect]:1, r:1, '=';
13    J,LOOPLIMIT←3;
14    JUDGE accept on line 10, col 14
15     [ans|[ok; pause for 1 sec; IF J,COUNT=1 THEN oks←oks+1]
16      l+r|[no; write " did you add?" for 1 sec];
17      l-r|[no; write " did you subtract?" for 1 sec];
18      l×r|[no; write " did you multiply?" for 1 sec];
19      l÷r|[no; write " did you divide?" for 1 sec];
20    ELSE
21     [IF J,COUNT=3 THEN write " Answer was ",ans:4 for 2 sec
22     ELSE [no; write " Try again" for 1 sec
23
24  erase; on line 20,col 5 write "Number correct =",oks,"
25  Number missed =",10-oks until [NEXT]
```

5. **Listing for a simple "HANGMAN" game program which requires that the player guess letters used to spell a hidden word. Each letter guessed which does not appear in the word results in parts of the man being drawn until he is "hung."**

```
Procedures: HANGMAN      Space: 553      >
1   (The classic HANGMAN game, guess the letters in a word)
2
3   CONSTANT
4    INTEGER numofwords←20, x←200, y←125;
5    ARRAY[0:numofwords] OF STRING[8] words←
6     ('simian', 'canoe', 'computer', 'seagull',
7      'mangrove', 'eloquent', 'camel', 'tortoise',
8      'building', 'duck', 'aircraft', 'sunshine',
9      'pumpkin', 'violent', 'erudite', 'swift',
10     'helpless', 'diligent', 'superior', 'beastly',
11     'gross');
12
13  VARIABLE
14   INTEGER i,right,missed,select;
15   SET OF 0:numofwords  used;
16   SET OF 'a':'z'  charsinwd, usedchars;
17   STRING[8] word;
```

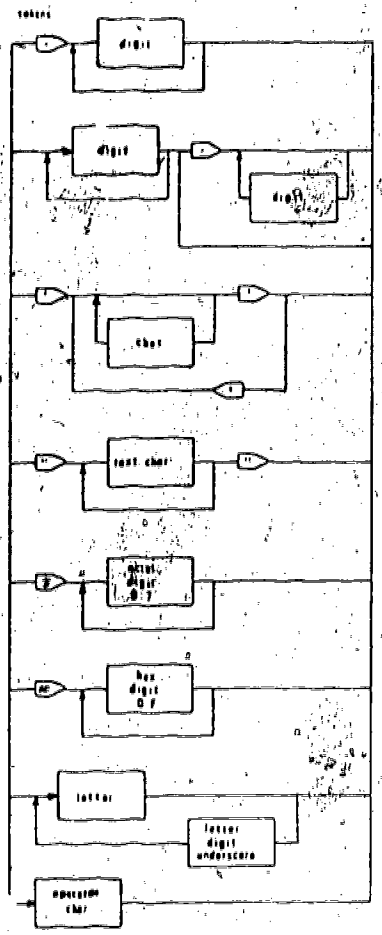
```
10
19  again:
20  erase; right←0; missed←0; charsinwd←[]; usedchars←[];
21  REPEAT numofwords×3 UNTIL ¬(select ∈ used) DO
22    select ← RANDOM × numofwords;
23  used←used + [select]; word←words[select];
24  FOR i TO LENGTH(word) DO charsinwd←charsinwd + [word[i]];
25  on line 4,col 5 sized 3.0 write large 'The Hangman Game';
26
27  (Draw the Gallows)
28  connect x+250,y,  x,y,  x,y-100, x+250,y-100,
29          x+250,y+300, x+55,y+300, x+55,y+275;
30  JUDGE accept rep with[noarrow,nocaps]
31    ['a':'z']
1     IF J,KEY ∈ usedchars THEN
2       [on line 25,col 10 for 1 sec write"You used that char";
3       J,FLAG←FALSE;
4     ELSE
5       [usedchars←usedchars+[J,KEY];
6       IF J,KEY ∈ charsinwd THEN
7         [FOR i TO LENGTH(word) DO
8           IF J,KEY=word[i] THEN
9             [write J,KEY on line 20, col(i+10); right←right+1];
10        IF right=LENGTH(word) THEN
11          [write "You win" for 3 sec until [NEXT]
12          on line 25, col 10; GOTO again
13        J,FLAG←FALSE
14      ELSE
15        [missed←missed+1;
16        CASE missed OF
17          [1] draw from x+5,y+5 to x+55,y+50; (left leg)
18          [2] draw from x+105,y+5 to x+55,y+50; (right leg)
19          [3] draw from x+55,y+50 to x+55,y+160; (trunk)
20          [4] draw from x+55,y+150 to x+5,y+100; (left arm)
21          [5] draw from x+55,y+150 to x+105,y+100; (right arm)
22          [6] circle 10 at x+55,y+180 eccentricity 2.0; (head)
23          [7] dots x+50,y+190, x+60,y+190; (eyes)
24          [8] dots x+55,y+185, x+54,y+184,
25                   x+55,y+184, x+56,y+184; (nose)
26          [9] connect x+51,y+170, x+53,y+168, x+57,y+168,
27                      x+59,y+170; (mouth)
28          [10]
29            [draw from x+55,y+275 to x+55,y+200;
30            on line 20, col 11 write word;
31            on line 25, col 10 write "You hung yourself"
1               for 3 sec until [NEXT];
2             GOTO again;
3
4         J,FLAG←FALSE;
5
6
7
8   ELSE
9     write "Not a letter" for 1 sec on line 25, col 10;
```

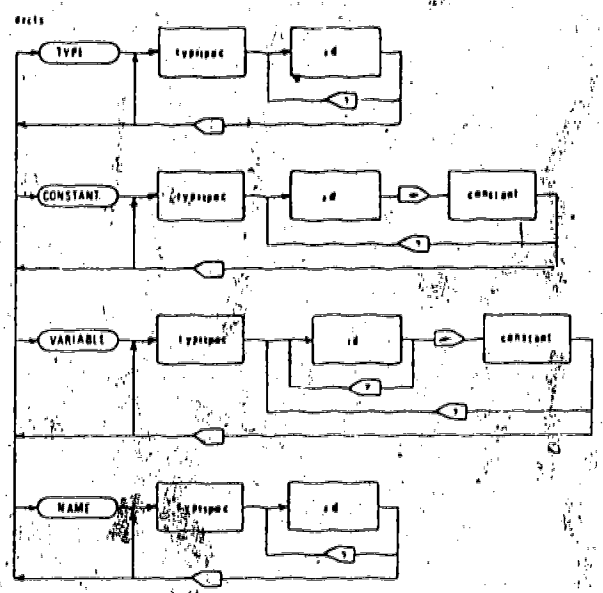
6. A single frame taken during the execution of the game. The user is trying to guess the word "beastly" but is very close to being "hung."

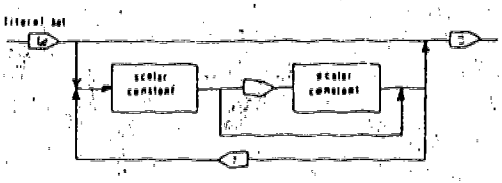


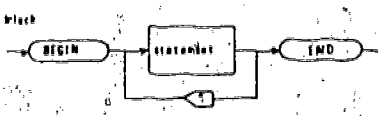
APPENDIX B; CAMIL II LANGUAGE SYNTAX CHARTS

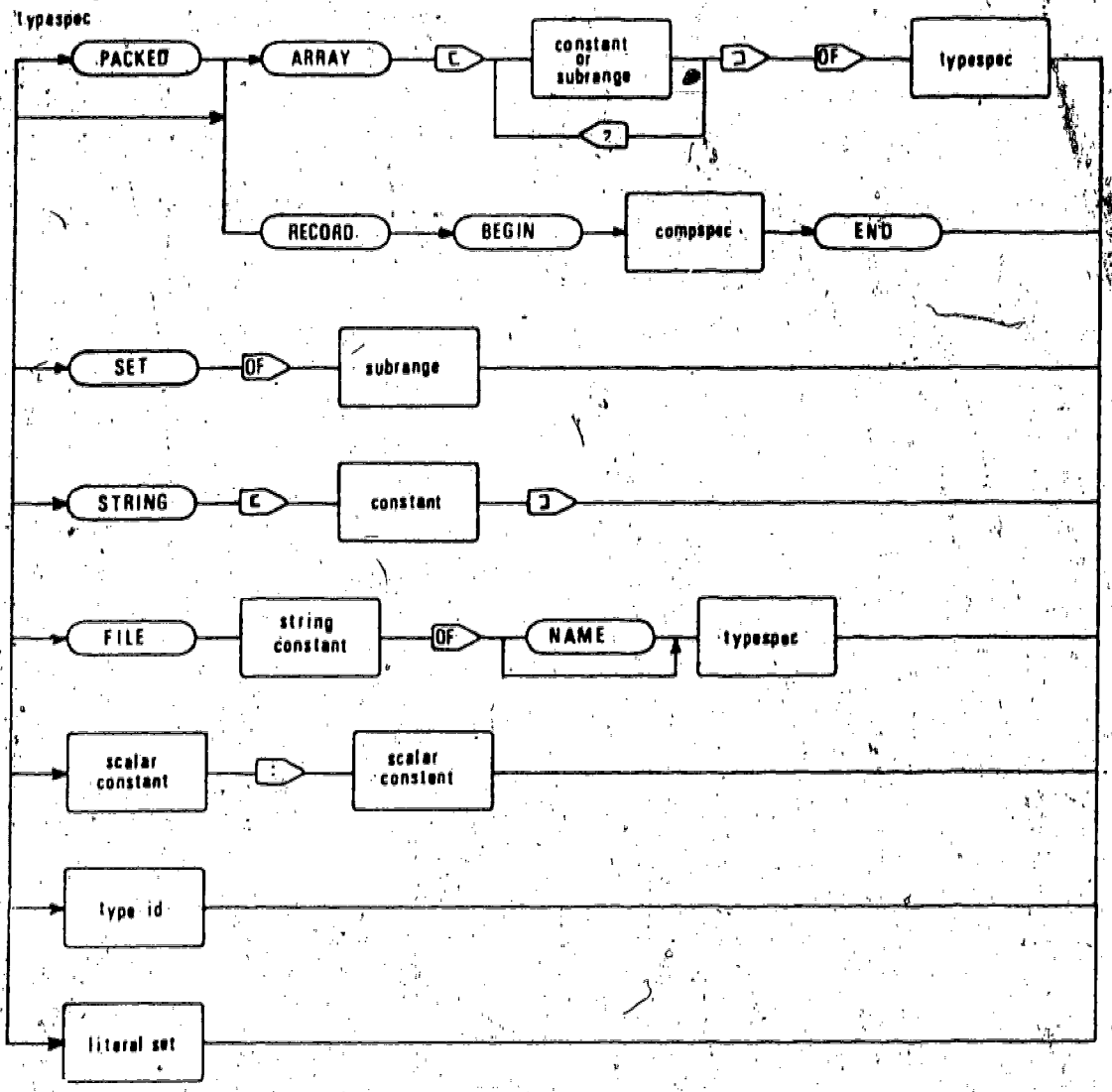
The following charts represent the syntax of the CAMIL II language graphically. The explanation of chart notation is included in the Language Description section of this report. The following charts do not necessarily explain semantic restrictions of the language, which are explained more fully throughout the report.

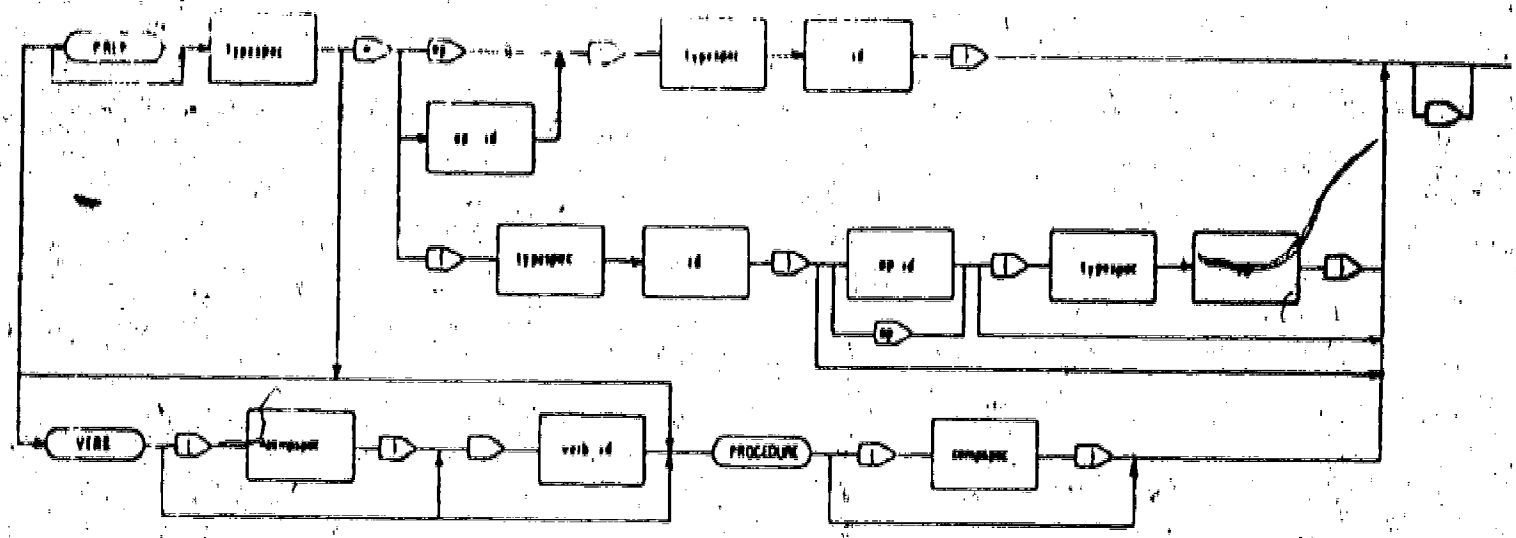


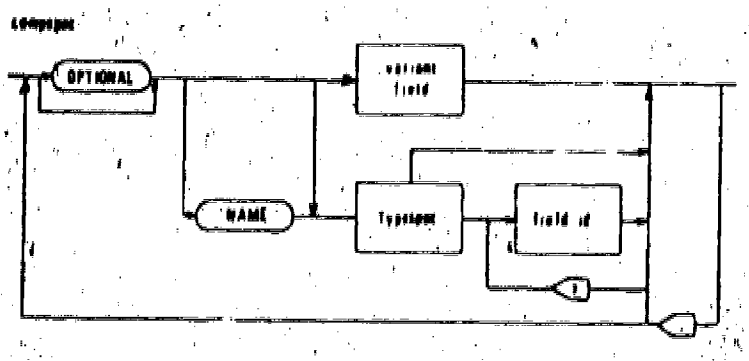


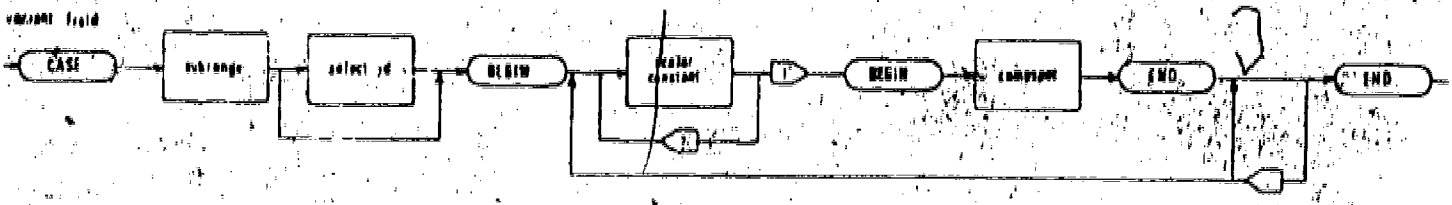


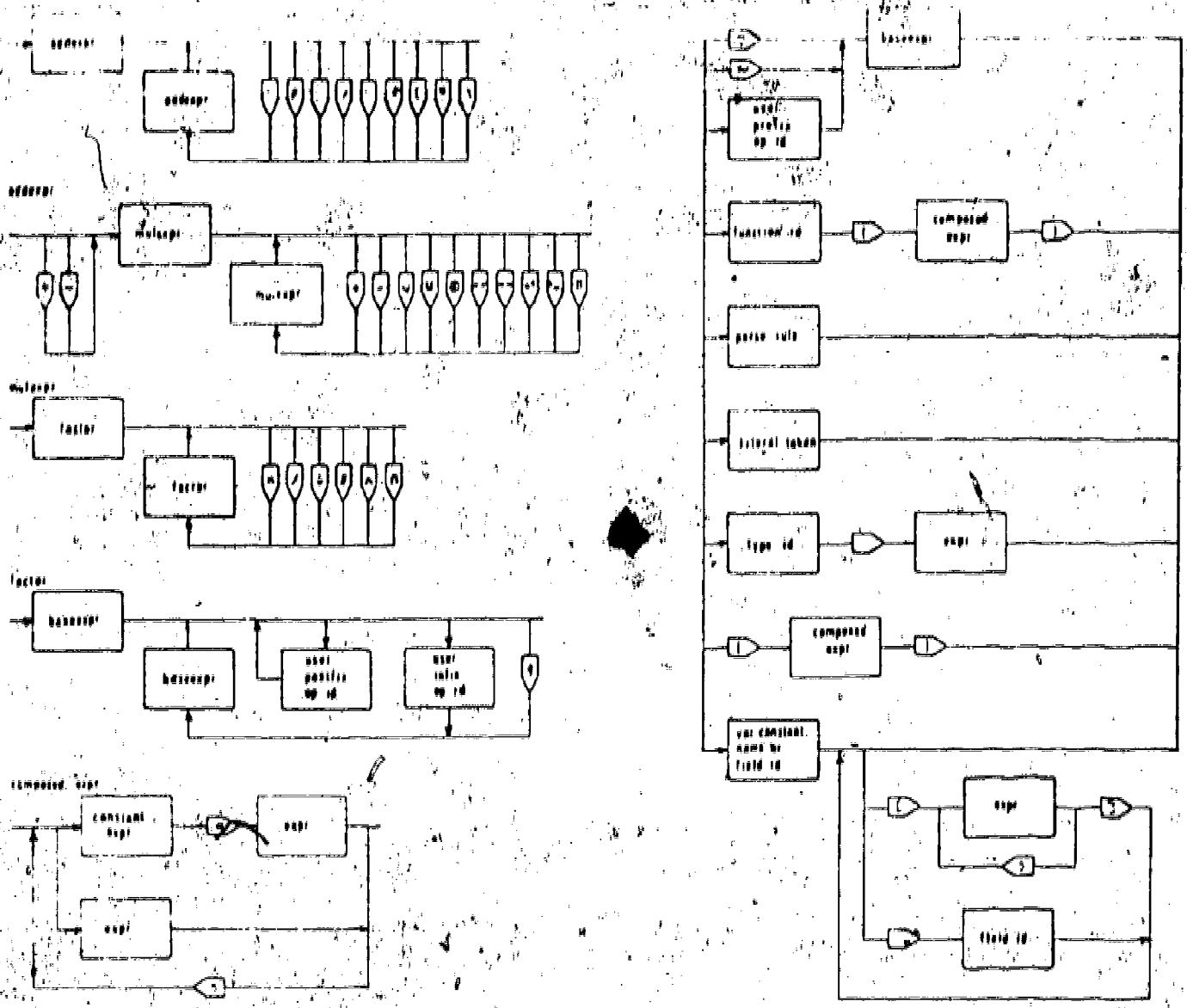


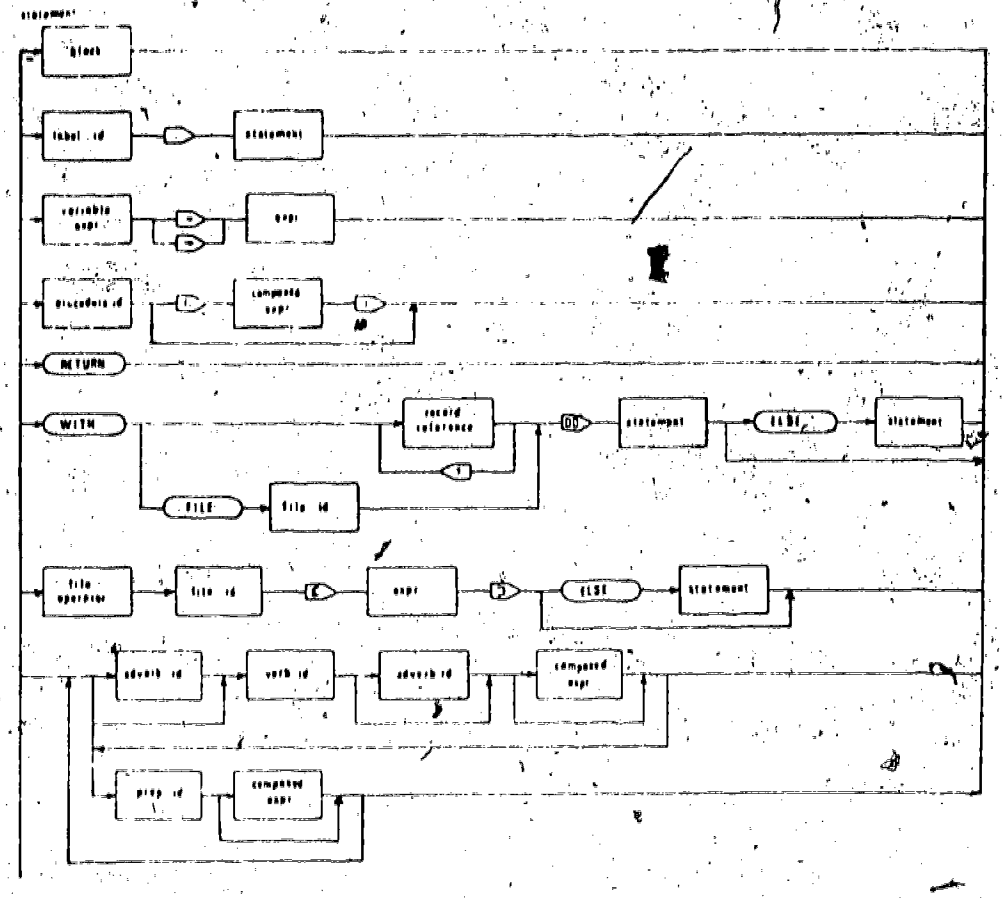

typespec
PACKED
ARRAY
[
constant or subrange
,
]
OF
typespec
RECORD
BEGIN
compspec
END
SET
OF
subrange
STRING
[
constant
]
FILE
string constant
OF
NAME
typespec
scalar constant
:
scalar constant
type id
literal set

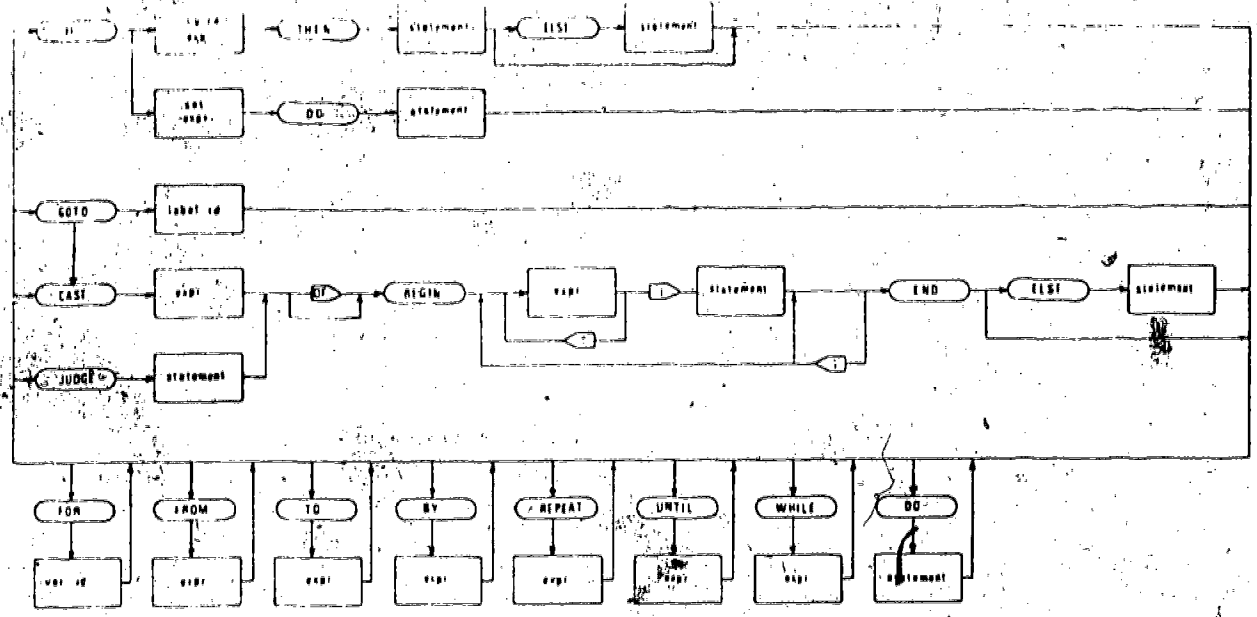

PROC
typespec
id
VERB
verb id
PROCEDURE


OPTIONAL
variant field
NAME
Typespec
field id


CASE
subrange
select id
BEGIN
BEGIN
END
END







☆U S GOVERNMENT PRINTING OFFICE: 1978- 671-056/76