OCAS - ON-LINE CRYPTANALYTIC AID SYSTEM

by

DANIEL JAMES EDWARDS

S.B., Massachusetts Institute of Technology
(1959)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
January 17, 1966

Signature of Author Daniel James Edwards
Department of Electrical Engineering, January 17, 1966

Certified by (Original signed by Marvin L. Minsky)
Thesis Supervisor

Accepted by (Original signed by Truman S. Gray)
Chairman, Departmental Committee on Graduate Students

This empty page was substituted for a blank page in the original document.

OCAS - ON-LINE CRYPTANALYTIC AID SYSTEM

by

DANIEL JAMES EDWARDS

Submitted to the Department of Electrical Engineering on January 17, 1966, in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

Deficiencies of various programming languages for dealing with quantities frequently encountered in cryptanalysis of simple cipher systems are discussed. A programming system is proposed which will permit a cryptanalyst to write and debug programs to aid in the solution of cryptograms or cryptographic systems. The basic elements of the proposed programming system are discussed in detail. They include: 1) a programming language to handle both algebraic quantities and character strings, 2) a display generator to permit quick specification of a display frame containing both alphanumeric strings and numerical data for an on-line CRT display device, and 3) an on-line program to control operation of the system and aid in debugging programs written in the proposed language.

Thesis Supervisor: Marvin Lee Minsky

Title: Professor of Electrical Engineering

ACKNOWLEDGEMENTS

The author would like to express his appreciation to Prof. Marvin L. Minsky who acted as thesis supervisor; to Messrs. Edward L. Glaser and Oliver G. Selfridge for their constant source of inspiration in completing this thesis; to Mr. Donald K. Pollock for his help in obtaining hard to get cryptographic literature; and to my wife Joyce for her patience during the entire project and help in typing the final manuscript.

TABLE OF CONTENTS

TABLE OF CONTENTS

APPENDICES PAGE

SECTION 1

DEFINITIONS

This thesis is primarily concerned with an on-line computer programming system designed to ease the work of a cryptanalyst.

The following definitions are given to acquaint the reader with some of the terms commonly encountered in the field of cryptanalysis.

Cryptology is the branch of knowledge that deals with the development and use of all forms of secret communication.

Cryptography is the branch of cryptology that deals with secret writing.

Cryptanalysis is the branch of cryptology that deals with the analysis and solution of cryptographic systems.

A Cipher is a cryptographic system which conceals, in a cryptographic sense, the letters or groups of letters in the message or plaintext. (Appendix A gives a list of common cipher systems.)

Enciphering is the operation of concealing a plaintext, and the result is a cipher text, or in general a cryptogram.

Deciphering is the process of discovering the secret meaning of a cipher text.

A key is the variable parameter of a cipher system, prearranged between correspondents, which determines the specific application of a general cipher sysem being used. The use of keys permits almost endless variations within a given cipher system. In fact, the value of a specific cipher system is based on how hard it is for an "enemy" to break a cryptogram or series of cryptograms, assuming he knows the complete details of the system but lacks the keys which were used to encipher the cryptograms originally. (See Appendix B for an example of a cipher and a key.)

A code is a cryptographic system which substitutes symbol groups for words, phrases, or sentences found in the plaintext. It involves the use of a codebook, copies of which are kept by each correspondent.

Encoding is the operation of concealing a message using a code.

Decoding is the process of recovering an encoded message.

A code differs from a cipher because a code deals with plaintext in variable size units, such as words or phrases, while a cipher deals with plaintext in fixed size units, usually a letter at a time.

This empty page was substituted for a blank page in the original document.

SECTION 2

INTRODUCTION

The history of using ciphers to convey messages from one person to another goes back to earliest times. The Scytale, a transposition cipher device, was originated by Lacedaemonians and used extensively in Cicero's time. Modern substitution ciphers can be traced back to the cipher used by Julius Caesar, who substituted D for A, E for B, F for C, ... to B for Z, etc. in correspondence that he wished to keep from prying eyes.

The invention of the printing press brought many people in contact with the field. The first of a nearly constant stream of books on cryptology was Chronologia Mystica, published by Trithemius, abbot of Spanheim and Wuerzburg, in 1516. Since that time much progress has been made in the use of ciphers and codes for diplomatic, military, and even criminal purposes. Books by Yardley and Pratt (see Bibliography) give graphic pictures of the uses of codes and ciphers from the middle ages up through the late 1920's.

By the late 1930's, advances in the art of communications made cryptology a very sensitive area. The first rumblings of World War II led the governments of the major world powers to impose an information blackout on new literature available on the general subject of cryptanalysis. Since then, no new major works on the subject have been made available to the general public (with the possible exception of EYRAUD's book - see Bibliography). All books published since 1940 have dealt with analysis of cryptographic systems which have been common knowledge since the late 1920's. Public interest in the field has been maintained by the American Cryptogram Association (ACA) which was founded in 1929 and still publishes The Cryptogram, a bi-monthly magazine of articles and cryptograms. The hobby of solving cryptograms provides a fascinating intellectual challenge to those so inclined. Patient analysis and flashes of insight, combined with the thrill of uncovering something hidden, give cryptanalysis an enjoyment which is almost unique.

The advent of modern high-speed digital computers raises speculation as how best to apply the computer's vast bookkeeping powers to the field of cryptanalysis. Cryptanalysis may be thought of as a recursive process where one forms hypotheses and then checks the validity of the resulting implications. And creativity is associated with the forming of new hypotheses. By rapidly and accurately checking the validity of implications, computers can

provide the analyst with information needed to form new hypotheses.

The kind of aid a computer would provide can be seen in Yardley's discussion of breaking the Japanese diplomatic code preceding the Washington Armament Conference of 1921-22. The clerical work in this instance required preparing 60,000 index cards with fragments of Japanese messages in both plain and code text. This preparation was done by a "corps of typists" working many hours. After the cards were prepared, they were sorted into various categories and summerized by hand onto large summary sheets. Tasks like this could easily be accomplished by a digital computer.

Solution of ciphers also requires a certain amount of routine bookkeeping, such as counting letter frequencies and looking for repeated digraphs. Also, Colonel Friedman's advice about using a soft pencil with a big eraser is well taken, for in solving cryptograms by hand the eraser is used almost as frequently as the pencil.

Let us again examine the idea of using a computer, this time with a CRT display. Why not have the computer allow an operator to make a guess and watch the computer work out the consequences? If the guess does not "prove out", the operator can erase the guess and its consequences with a single key stroke. The advent of modern time-shared computer systems, complete with CRT displays, places all of the above-conjectured uses of a computer within the realm of practicability, because an expensive computer need not be tied up while the analyst is trying to figure out what to do next.

The problem then resolves to: what language can a cryptanalyst use to program an on-line computer to perform the various tasks pertaining to solving a cryptogram? Let us list some of the requirements for such an On-line Cryptanalysis Aid Language (OCAL) and then examine some existing languages in light of these requirements. First, the OCAL must handle strings of alphabetic characters and manipulate these strings easily. Second, the OCAL must handle algebra with ease, including matrix operations. Third, the OCAL should be embedded in a machine environment which allows the cryptanalyst to write and check out programs on-line. Finally, the OCAL must be reasonably efficient in its use of computer time and storage, if reasonable response times are desired in a time-shared computer environment.

Available languages for programming computers include basic machine language, LISP and its derivatives, the ALGOL family of languages, and string-processing languages such as METEOR and SNOBOL. Machine language, even with macros, is rejected because it is much too hard to program and quickly check ideas. The OCAL should be a tool which a cryptanalyst can use easily, while machine

language, even in the hands of a skilled programmer, is a blunt instrument at best. LISP on the other hand, while not easy to learn, is a powerful language for many types of complex data manipulation tasks. LISP handles algebraic tasks with moderate ease, matrix manipulations with some difficulty, and strings with still more difficulty. Finally, storage efficiency leaves much to be desired, and this objection is especially critical when the problem of using large dictionaries in the OCAL is considered. Therefore, LISP is rejected as the OCAL. The other LISP-like languages, such as SLIP, threaded lists, and IPL (the machine language of list processing) suffer similar deficiencies.

Next, the ALGOL family of languages, such as ALGOL, MAD, AED, PL/I, and even FORTRAN is considered. These languages handle algebra with ease, but their string-handling abilities are almost non-existent. Furthermore, none of these languages is particularly well adapted to on-line use. This, coupled with the difficulty of adding good string-processing features to any current time-sharing version, leads us to look elsewhere for the OCAL.

Finally, let us examine the rather interesting string-processing language SNOBOL. The heart of SNOBOL is an elegant pattern-matching algorithm which allows a string to be tested for a complicated pattern in one statement. In order to test the suitability of SNOBOL for cryptanalysis, a program to solve the simple railfence cipher was written and debugged in about 15 man-hours using the Compatible Time-Sharing System at Project MAC. (See Appendix B for a discussion of the railfence cipher and a resulting SNOBOL program.)

Writing the railfence program revealed several weaknesses in SNOBOL. First, the arithmetic was workable but somewhat awkward. Second, there was no provision for arrays, which made the solution scoring by digraphs rather difficult. This problem was solved in the railfence program by making the digraph scoring array into a series of fixed strings which were accessed by the pattern-matching statement.

The most serious deficiency of SNOBOL was the lack of a functional argument provision in the pattern-matching statement. That is, pattern elements could be fixed strings, arbitrary strings, arbitrary strings of fixed length, or repeats of previously-matched pattern elements. Missing was provision for making a pattern element into an arbitrary string, subject to a predicate procedure which could examine the state of the pattern match to that point. (This deficiency is not present in the string-processing language METEOR which is an improved LISP implementation of the string-processing language COMIT. However, METEOR still suffers

from the same problems as LISP, regarding efficient use of time and storage.)

These deficiencies ruled out SNOBOL as the OCAL, but the pattern-matching concept was considered important and was extended along the lines of allowing a pattern element to specify a predicate procedure. This extended SNOBOL statement was then incorporated in the final design of OCAL.

With no single language suitable for the OCAL, two courses of action were open. Either take an existing language and extend it to overcome deficiences, or design a new language aimed specifically at the field of cryptanalysis. The first alternative was rejected, because extending an existing language does not usually allow one to insert new ideas without redesigning the entire language. The author was interested in what could be done from scratch, and therefore he chose the second alternative, design of a new language.

Hence, the specific goal of this thesis is to specify and demonstrate an On-line Cryptanalytic Aid System (OCAS) which will permit a computer programmer, who is already familiar with cryptanalytic procedures, to easily program and test an attack on any of the 30 different cipher systems that regularly appear in The Cryptogram (Again, see Appendix A).

SECTION 3

ON-LINE CRYPTANALYTIC AID SYSTEM (OCAS)

The proposed On-line Cryptanalytic Aid System (OCAS) has the following parts. First, a computer programming language, OCAL, which easily handles both algebraic calculations and character string manipulations. Second, an On-line Cryptanalytic Display Generator (OCDIS) to allow people to interact more rapidly with the program than using just a teletypewriter. And finally, an On-line Debugging and Control Program (ODBUG).

Each of these parts will be discussed in later sections. This section will discuss some of the basic design criteria of the OCAS. First, the system should be reasonably easy to use, once the basic languages involved are learned. Second, the system should contain a complete set of text-editing and program-debugging aids. Third, the system should be "fail-soft". That is, it should be forgiving to common programming mistakes and the operator should be able to regain control of a run-away program. Finally, the system should be open-ended so that new programs can be added with ease. And using OCAS, the cryptanalyst should be able to let the computer handle most of the bookkeeping tasks involved in solving a cryptogram or cryptographic system.

Another design criterion for the OCAS was ease of implementation. The difficulties of fully implementing ALGOL are well known. Everything in OCAS had to be easily implementable on a reasonable machine. It was hoped that a skeletal implementation of OCAS could be completed in four months. This introduced the conflicting design goals of a complete language versus speed of implementation. During development of system specifications, this conflict was usually resolved in favor of a complete language: so as a consequence, the skeleton implementation was started but not completed.

3.1 ON-LINE CRYPTANALYTIC AID LANGUAGE (OCAL)

OCAL is a problem-oriented computer programming language with the general area of cryptanalysis as the problem domain. OCAL is basically a synthesis of the MAD and SNOBOL computer programming languages, combined with ideas taken from SLIP and PL/I. This section describes the basic features of OCAL. (A complete description of OCAL syntax can be found in Appendix C.)

3.1.1 Basic Data Types

The following quantities comprise the OCAL basic data types:

a) Logic - a two-bit quantity standing for True, False, Neither, or Undefined. The reason for introducing a basic four-value logic is to make the results of certain logical comparisons more obvious to the programmer. For instance, the question "Is ten greater than an orange?" could be answered "Undefined" because the quantities involved in the comparison are not comparable. An example of use for logic value "Neither" might be in response to the question "Given that cipher A stands for plaintext Q in a simple substitution cipher, does cipher text MKP stand for plaintext THE?" The answer "Neither" in this case means undecided, for the information given is insufficent.

Situations requiring a simple Boolean decision can be made on a "True" or "Not True" (e.g., "False", "Neither", or "Undefined") basis.

b) Integer - the standard computer quantity used for integer arithmetic and subscripting expressions for compound data structures.

c) Real - floating-point numbers used primarily in arithmetic calculations.

d) Character - a two- to eight-bit representation of a member of the ASCII character set. Each character is associated with an alphabet (defined next) which gives the mapping from a particular ASCII character subset into the full ASCII character set. The Character is the basic constituent of the string (defined later) and may also be used in subscripting expressions for compound data structures.

e) String - an ordered set of characters all taken from the same alphabet. A string may be arbitrarily long and is associated with an alphabet that gives the mapping of character representations into ASCII characters. Also associated with a string is an integer giving its current length in characters.

f) Reader - an object which may be associated with a string. A reader may be thought of as the reading head of a Turing machine, with the associated string being the Turing-machine tape. A reader can move up and down a string, read characters out, or write characters into a string. In addition, a reader can be positioned

at the head of a string, at a preset place on the string, or at an arbitrary place on the string.

g) Alphabet - defines a mapping function from the ASCII character set (the standard OCAL alphabet) into a subset of ASCII. The alphabet concept is used to gain storage and subscripting efficiency when dealing with characters and strings. An alphabet may map any number of characters in the domain (ASCII) into a single character in the range. Characters appearing in the domain, but not in the range, are mapped into the null character (i.e., ignored). In addition, each alphabet provides two extra characters in the range corresponding to logic values "Neither" and "Undefined". This feature allows OCAL to indicate certain logical decisions or conditions within a string.

Also associated with each alphabet is an integer equal to the cardinality of the mapping range, excluding the logical characters "Neither" and "Undefined". This permits character and string arith-metic to be done modulo the size of the alphabet.

h) Statement Label - a special data type referring to a part of an OCAL procedure. Statement labels are data types to permit assigned GO TO statements and functional arguments in OCAL.

3.1.2 Compound Data Structures

The OCAL compound data structure is taken from the PL/I language. Compound data structures can consist of any of the previously-mentioned basic data types and other compound data structures. Various parts of a compound data structure can be accessed either by name or by subscripting expressions. Thus, a real array in OCAL is simply an n-dimensional compound data structure consisting of real numbers.

3.1.3 Declarations

Declarations are used in OCAL to associate data types with the local variables used in a procedure. All variables must be declared at the head of the procedure or block in which they appear. Variables may be either local or global in scope: local variables are defined only within the block or procedure containing the declaration, and global variables are defined in all blocks and procedures.

Declarations are also used to define compound data structures; in which case all the elements of the declaration must be basic data types or already-declared compound data structures. That is, recursive definition of a compound data structure is not permitted.

3.1.4 Statements

Statements in OCAL may be either simple or compound. Simple statements are terminated by a semi-colon, or the end of the line on which they appear, unless the continuation character "." (period) appears as the first character on the following line. Executable statements may be symbolically labeled with one or more labels.

Compound statements are groups of statements enclosed within the statement parentheses, BEGIN and END. A compound statement is called a block, and blocks may be nested to any depth.

OCAL statements fall into the following categories:

a) Declarations - type identification, data structure, and procedure structure;
b) Control - GO TO, conditional, and iteration;
c) String pattern matching - similar to the basic SNOBOL string pattern-matching statement;
d) Assignment - assigns values to symbolic quantites;
e) Execute - calls a specific procedure, but ignores any values returned;
f) Error control - allows an OCAL procedure to retain control even though a called procedure has taken an error return.

(A detailed list of statements with their syntax is in Appendix C.)

3.1.5 Procedures

Procedures may have a fixed or variable number of arguments or parameters. If the procedure has a variable number of parameters, the global integer variable "NUMBEROFPSETS" gives the number of parameter sets for any particular procedure call. Parameters are referenced by the local name which is given procedure declaration.

Procedures may be defined recursively and keep their working storage on push-down lists. Procedure calls are made in the form

fn.(a1,a2, ...,an)

where "fn" is the procedure name and the period [.] distinguishes a procedure call from a subscripted variable. The "ai"s are the parameters for the called procedure.

A procedure with no arguments is called by the procedure name followed by a period.

A procedure may be given a value by the statement

VALUE e

where "e" is any expression.

There are two procedure returns in OCAL; first, the normal return is specified by the statement

RETURN e

or by executing the last statement of a procedure, and the second return is given by the statement

ERROR s

where "s" is a string. On executing an error return, control is returned to the last procedure which executed the statement

ON ERROR, s

where "s" is any simple or compound statement (usually a GO TO statement, or a block ending with a GO TO or DISMISS statement).

3.1.6 Relations

These are logical operators that compare integer, real, character, and logical quantities. The value of a comparison is the logical quantity "True" if the relation holds, "False" if the relation does not hold, and "Undefined" if the quantities are incomparable (e.g., is "blue" equal to 3.14?).

3.1.7 Arithmetic

Normal infix operators may be used in arithmetic expressions in OCAL. Each operator takes operands whose type is character, integer, or real and produces a result which is the same type as the highest type of any operand; the ranking between types is character lowest, integer next, and real highest. Furthermore, if characters appear in any arithmetic expression, the result is taken modulo the alphabet size associated with the first-mentioned character. This feature may be suppressed if desired.

3.1.8 Logical Expressions

Standard logical infix operators are available in OCAL. Each operator takes two arguments whose type is logic, character, or integer. The logical operators produce a result which is the same type as the highest types of any operands; the types being ranked with logic lowest, character next, and integer highest. The value of a logical operator is the bit-wise combination of the operands after type transfers (if any) have been performed.

3.2 ON-LINE CRYPTANALYTIC DISPLAY GENERATOR (OCDIS)

OCDIS is intended to permit a cryptanalyst to easily specify a CRT display of the quantities available in OCAS. The display may either be fixed (unchanged until the cryptanalyst interacts with the computer) or dynamic (changed periodically to reflect intermediate results associated with some continuing set of procedures being executed by the computer). The display is organized using a set of formats which correspond to pages in a book. The operator can

"flip" pages with pushbutton commands from the display console. A static display is compiled once, each time it is brought onto the screen. A dynamic display is compiled when brought onto the screen and then portions of it are periodically recompiled to keep up with changing portions of input data.

The CRT display itself is run in program-interrupt mode, so that computations can proceed even when a display is visible. (A summary of the features of OCDIS is given next, and detailed specifications for the OCDIS programs may be found in Appendix D.)

Each display contains a log in the upper left hand corner which gives the current date, time, frame number, and title for identification of still photographs taken of the display. This log is maintained by the system and thus is not a burden to the cryptanalyst.

The main data type used in a cryptographic display is the string. A string display may be organized by the number of strings to be displayed in parallel. For a simple substitution display this could be three lines; one for the cipher text, one for the plaintext, and one blank line for general eye relief. The line length in characters is preset, and when data is supplied this basic three-line format is repeated down the display until all data are used.

In addition to strings, OCDIS can display other basic data types and compound data structures, such as matrices and character arrays. Vectors can be displayed either as a table of numbers or as a bar graph.

3.3 ON-LINE DEBUGGING AND CONTROL PROGRAM (ODBUG)

ODBUG is similar to the DDT family of debugging packages for the Digital Equipment Corporation PDP-1,4,5,6, and 8 computers. ODBUG permits the cryptanalyst to examine and set the contents of variables. It can also execute OCAL statements interpretively, and thus acts as the OCAS control program by calling the various OCAL programs the cryptanalyst wants to use.

ODBUG can also be used to set break-points in OCAL programs which, when executed, will return control to ODBUG. If the analyst is satisfied with the program's performance, he can resume the program at the break-point or he can initiate another procedure.

Since OCAS is adept at handling strings, and since an OCAL program is basically an ASCII string until it is compiled, ODBUG can call procedures to perform simple editing functions on OCAL programs that are stored as strings. Thus, with ODBUG as a control program, OCAS will be a complete system for writing, editing, debugging, and running programs written in OCAL. (Complete specifications for ODBUG can be found in Appendix E.)

SECTION 4

IMPLEMENTATION

The initial implementation of OCAS will be as an interpreter for the Digital Equipment Corporation (DEC) PDP-6 computer, using the DEC Type 340 display located at Project MAC. This computer is run by the Project MAC Artifical Intelligence Group, under the direction of Prof. Marvin L. Minsky, and has many advantages for on-line experimentation with systems using computer-generated displays.

The SNOBOL-type, string pattern-matching algorithm, a simple storage-control algorithm, and the elementary reader functions have already been programmed in PDP-6 machine language. The next step is to program the OCAL interpreter in machine language. After that, input/output procedures will be programmed around the standard PDP-6 input/output package for the on-line teletype, paper tape reader, paper tape punch, and DECtape unit. Finally, the basic OCDIS and ODBUG routines will be programmed in machine language. It is estimated that this first implementation of OCAS will take from 500 to 1000 man-hours to program and check out.

After experience is gained with OCAS in an interpreted form, an OCAL compiler and loader can be written to increase the efficiency of debugged OCAL programs. Specifying, programming, and debugging this package will take an additional 1000 man-hours.

This empty page was substituted for a blank page in the original document.

SECTION 5

EXAMPLES OF OCAL APPLIED TO CRYPTOGRAPHIC PROCESSES

So far, the design of OCAS has been based on the author's intuition of what he would like to have the computer do as an aid to solving cryptograms. This intuition is based both on experience with inter-active computer systems, and with solving ACA cryptograms in several cryptographic systems using pencil and paper. Let us examine some of the elementary cryptographic bookkeeping tasks and see how these would be expressed in OCAL. (An example of a complete OCAL procedure to find the period of a periodic cipher, such as a Vigenere or Beaufort, can be found in Appendix F.)

5.1 SIMPLE FREQUENCY COUNT

Often a count is made to determine how many times each letter is contained in a cryptogram. To do this kind of count in OCAL would require the declarations:

```
CHARACTER C
READER SCAN
ALPHABET ENG('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
STRING CRYPT
INTEGER FCOUNT
DECLARE FCOUNT(ENG)
```

[The last declaration makes FCOUNT a vector equal in length to the alphabet ENG, which contains just the letters A through Z.]

```
CRYPT = READ.('PTR','.')
```

[Read an ASCII string from the photoelectric paper tape reader, up to and including the first period.]

```
CRYPT = ENG.(CRYPT)
```

[Convert the string into the alphabet ENG.]

```
ATTACH.(SCAN,CRYPT)
```

[Attach the reader SCAN to the string CRYPT.]

```
C = $C.(SCAN)
ENDSTRING = F!
DO UNTIL ENDSTRING, BEGIN
FCOUNT(C) = FCOUNT(C)+1
```

[Enter a DO loop with character variable C set to the first character of the string CRYPT.]

```
C = $IC.(SCAN)
END
```

The reader function $IC. advances the reader one character position and reads the next character into the variable C. When the reader reaches the end of the string CRYPT, the frequency count will be found in the vector FCOUNT.

5.2 LONGEST REPEATED SEQUENCE

The problem here is to find the longest, non-overlapping, repeated sequence in the string CRYPT. This example demonstrates the SNOBOL-type pattern-matching statement in OCAL:

```
INTEGER N
STRING CRYPT,FILL,RPT,R
N = 1
```

[Set the length of the first trial repeated string to one.]

```
SCANFLAG = T!
DO WHILE SCANFLAG, BEGIN
CRYPT *R/N* *FILL* R
```

[Scan the string CRYPT for the first instance of a string of length N followed by an arbitrary string, followed by a repeat of the first string. If a match is found, set the string variables R and FILL to the substrings of CRYPT that they match.]

```
N = N+1
RPT = R
END
```

When the SNOBOL pattern scan succeeds in finding a match, the length of the trial string is incremented by one and the repeated string found on this trial is remembered in the string RPT. When the DO loop terminates, the first occurrence of the longest non-overlapping repeated string will be found in RPT.

SECTION 6

CONCLUSIONS

This thesis describes an inter-active computer programming system (OCAS) which is intended to ease the solving of cryptograms by giving a cryptanalyst the necessary tools to easily program a computer. As may be expected in this type of project, the system has grown considerably since its inception. Unfortunately, it was not possible to completely program and debug the described system in the time available for this thesis.

A computer programmer who is working with cryptographic systems frequently deals with both character strings and algebraic quantities. The programming system described has a computer programming language (OCAL) which is intended to manipulate both of these kinds of data. Note, however, that OCAL is definitely a language for computer programmers who are familar with cryptographic procedures: it is not an attempt to produce a "COBOL" for cryptanalysis.

The programming system also includes a display generator to permit easy specification of CRT displays to accompany OCAL programs. In addition, the programming system includes an on-line debugging and control program (ODBUG) to ease debugging of programs written in OCAL.

Even though the system described was intended to provide computer aid for solving cryptograms, certain parts of the system may be of interest to people designing other inter-active computer software. The concepts to be emphasized in this respect are: first, the general "fail-soft" design philosophy of not letting innocent programming mistakes "bring the house down", and second, the integrated system of program writing, editing, debugging, and running.

The other portion of this thesis which may be of interest to persons not interested in the field of cryptanalysis is a discussion of deficiencies in the SNOBOL string-manipulation language. For example, addition of predicate procedures to the SNOBOL scan algorithm greatly enhances the power of the language. This additional power may be of use to those interested in natural-language processing.

This empty page was substituted for a blank page in the original document.

APPENDIX A

COMMON CIPHER SYSTEMS

Problems enciphered in the following cipher systems appear regularly in the bi-monthly magazine THE CRYPTOGRAM published by the American Cryptogram Association. This list is included to show the variety of cipher systems that people frequently solve with pencil and paper. It has been the goal of this thesis to write a computer programming language which will permit a cryptanalyst to attack any one of these systems quickly and easily.

Amsco
Beaufort
Beaufort, Variant
Bifid
Cadenus
Fractionated Morse
Grandpre
Grille
Gronsfeld
Keyphrase
Myskowsky
Nihilist Substitution
Nihilist Transposition
Phillips
Playfair
Playfair, Seriated
Porta
Portax
Quagmire (Vigenere with mixed tableaus)
Ragbaby
Railfence
Simple Substitution
Slidefair
Transposition, Auto
Transposition, Columnar
Transposition, Route
Tri-Digital
Trifid
Tri-Square
Vigenere
Vigenere, Auto Key
Vigenere, Running Key

This empty page was substituted for a blank page in the original document.

APPENDIX B

SOLUTION OF THE RAILFENCE CIPHER IN SNOBOL

The Railfence cipher is a simple form of transposition cipher. The plaintext is written in a zig-zag route thus:

```
S     R     N     H
 A   E A   E C   P E
  M L   I F   E I   R
   P     L     C
```

The cipher text is then taken off in rows giving the following cryptogram:

SRNHA EAECP EMLIF EIRPL C.

The key consists of how many letters deep the zig-zag is (known as the rail depth) and whether the zig-zag starts off in W form (as the example does) or in inverted W form.

The railfence program was written to test SNOBOL's suitability as a computer programming language for solving cryptograms. The method of attack used in the railfence program was to prepare a string as long as the cryptogram in the form:

1234543212345432123...

where the highest number in the string indicates the rail depth being tested. The letters of the cryptogram are taken off one by one and substituted for the 1's first, then the 2's, etc. The resulting string is the trial decipherment which is then scored by using a digraph weight table. This table is found in strings CWA through CWZ in the program. The score is the sum of the weights for each digraph. In making a sequence of trials, the one with the highest score is chosen as the best solution for the cryptogram.

The program in SNOBOL was written to allow the operator to direct the search among the various rail depths and forms by asking for instructions. The instructions consist of simple entries indicating a particular depth and form, trials over a series of depths in both forms, or END which terminates the program.

Writing the program in SNOBOL revealed some serious limitations of the overall suitability of SNOBOL for general cryptographic work. Among these are: 1) lack of arrays and floating-point numbers, 2) lack of generalized functional arguments for the string pattern-matching elements, and 3) overall system slowness in running relatively simple examples. (The total main-frame computer time was

often more than 30 seconds even for simple examples).

The railfence program example is included here to demonstrate the kind of problem OCAL was designed to solve. In the example, bold capital letters are the computer program typing and lower-case letters are operator responses. Explanatory remarks added for this manuscript are set off by brackets.

AN EXAMPLE OF THE RAILFENCE PROGRAM

```
snobol rfence        [calling the program in CTSS]
W 1314.2             [time of day furnished by CTSS]
EXECUTION            [CTSS indication that the program is operating]

WHICH PROBLEM        [railfence program asking for problem ident.]
(m-a 63)             [problem found in the March - April 1963
                      issue of The Cryptogram]
PROBLEM IS
SASEP AISNI CRPOB INGAF COEAH OCNNR NSOIS OIIAH RT. (M-A 63)


TRY                  [railfence program asking for directions]
help                 [operator asking for directions]
OPTIONS ARE
ALL M THRU N
WHERE N AND M ARE POSITIVE INTEGERS INDICATING
THE INCLUSIVE RAIL DEPTHS TO BE TESTED BOTH IN
STRAIGHT AND INVERTED W STYLE,
END
WHICH TERMINATES THE RUN,
+N
-N
FOR A SINGLE TRIAL
WHERE N IS THE NUMBER OF RAILS AND
THE SIGN INDICATES STRAIGHT OR
INVERTED W STYLE RAILS.

TRY
all 4 thru 6
RAIL DEPTH   SCORE
     +4        37
     -4        39
     +5        60
     -5        54
     +6        53
     -6        50

SOLUTION
SCORE ON 5 W STYLE RAILS IS 60
SIGNINASANFRANCISCOSHOEREPAIRSHOPBOOTICIAN

TRY
end                  [operator is satisfied with result and
                      terminates the run]
R 15.466+5.083       [15 seconds of computer time were spent
                      running the program and 5 seconds were spent
                      swapping it in and out of core. Statistics
                      are from an actual CTSS run]
```

```
*               RAIL FENCE PROGRAM
*
START           CWA = 'B2C2D1F1G1I1J1K1L2M1N3P2R2S2T2V2W1X1Y1'
                CWB = 'A1B1E4I1L2O1R1U2Y1'
                CWC = 'A2E2H2I1K2L1O2R1T1U1'
                CWD = 'A2D1E2F1I2J1O1R1U1V1Y1'
                CWE = 'A2C1D2F1G1L1M1N2P1Q2R2S2V1W1X3'
                CWF = 'A1E1F1I1L1O2R1T1U1'
                CWG = 'A1E2H2I1L1O2R2U1'
                CWH = 'A3E4I2O2T1'
                CWI = 'A1B1C2D1E1F2G1K2L2M1N3O2R1S2T2V2X2'
                CWJ = 'A1E1O2U1'
                CWK = 'A1E3I2N1O1S1Y1'
                CWL = 'A2D2E3I2K1L2O2S1T1U1Y3'
                CWM = 'A3B1E3I2M1O2P1U1'
                CWN = 'A1C1D2E1G2I1J1K1N1O1S1T2U1V1X1Y1'
                CWO = 'A1B1C1D1F3G1I1K2L1M2N3O1P2R2S1T2U3V2W2X1'
                CWP = 'A2E2H1I1L2O2R2T1U1'
                CWQ = 'U5'
                CWR = 'A1D1E2G1I1K1L1M1O1T1U1V1'
                CWS = 'A2E2H2I2K1L1M1N1O2P2Q1S1T3U2W1'
                CWT = 'A2E2H4I3O3R1S1T2U2W1'
                CWU = 'A1B1C1D1E1G1L1M1N2P1R2S2T2'
                CWV = 'A2E4I2O2'
                CWW = 'A2E2H3I2O1'
                CWX = 'A1C1E1I1O1P1T1'
                CWY = 'A1E1O2'
                CWZ = 'E3O2'
                MAXRAILS = '9'
                WRFLX('WHICH PROBLEM')
                IDA = RDFLX()
                ID = TRIM(IDA)
RD              HOLD = ''
RDA             SYSPIT *LINE*
                LINE 'END OF PROBLEMS'          /F(RDB)
                WRFLX('PROBLEM NOT FOUND')      /(END)
RDB             LINE ID         /S(RDC)
                LINE '.'        /S(RD)
                HOLD = HOLD LINE                /(RDA)
RDC             WRFLX('PROBLEM IS')
                HOLD *(C)*      /F(RDD)
                WRFLX(HOLD)
RDD             WRFLX(LINE)
                LINE = HOLD LINE
                LINE '.'        /S(RD1)
                HOLD = LINE
                SYSPIT *LINE*   /(RDD)
RD1             LINE *(C)* =
                EQUALS(C,'.')   /S(DN)
                EQUALS(C,' ')   /S(RD1)
                CIPHER = CIPHER C               /(RD1)
DN              WORK = CIPHER
                COUNT = SIZE(WORK)
TRY             WRFLX('')
                WRFLX('TRY')
                GUESS = RDFLX()
                GUESS = TRIM(GUESS)
                DELTA = '1'
                OSC = '0'
                N = '1'
                GUESS 'ALL' *F1* ' ' *N1* ' ' *F2* ' ' *N2*   /S(MLPA)
TRY1            GUESS '+' =
                GUESS 'END'     /S(END)
                GUESS '-' =     /F(DNA)
                DELTA = '0' - DELTA
                N = GUESS
DNA             .NUM(GUESS)     /S(DNB)
```

```
                WRFLX('OPTIONS ARE')
                WRFLX('ALL N THRU M')
                WRFLX('WHERE N AND M ARE POSITIVE INTEGERS INDICATING')
                WRFLX('THE INCLUSIVE RAIL DEPTHS TO BE TESTED BOTH IN')
                WRFLX('STRAIGHT AND INVERTED W STYLE.')
                WRFLX('END')
                WRFLX('WHICH TERMINATES THE RUN,')
                WRFLX('+N')
                WRFLX('-N')
                WRFLX('FOR A SINGLE TRIAL')
                WRFLX('WHERE N IS THE NUMBER OF RAILS AND')
                WRFLX('THE SIGN INDICATES STRAIGHT OR')
                WRFLX('INVERTED W STYLE RAILS.')                    /(TRY)
DNB             RAILS = GUESS   /(MLP2)
MLPA            .GT(N1,'1')     /F(TRY1)
                .LT(N1,N2)      /F(TRY1)
                RAILS = N1 - '1'
                MAXRAILS = N2
                WRFLX('RAIL DEPTH   SCORE')
MLP             RAILS = RAILS + '1'
                .GT(RAILS,MAXRAILS)             /S(FINIS)
                N = '1'
MLP2            PATTERN =
                D = DELTA
                C = COUNT
MLP1            PATTERN = PATTERN N
                N = N + D
                C = C - '1'
                .EQ(C,'0')      /S(PI)
                .GE(N,RAILS)    /S(REV)
                .LE(N,'1')      /F(MLP1)
REV             D = '0' - D     /(MLP1)
PI              WORK = CIPHER
                N = '1'
MOR             WORK *CH/'1'* =
MR1             PATTERN N = CH /S(MOR)
                N = N + '1'
                .GT(N,RAILS)    /F(MR1)
                NSC = '0'
                WORK = PATTERN
                WORK *SC/'1'* =
SL1             FC = SC
                WORK *SC/'1'* =                 /F(FIN)
                PNT = 'CW' FC
                $PNT SC *W/'1'*                 /F(SL1)
                NSC = NSC + W   /(SL1)
FIN             .GT(NSC,OSC)    /F(AGN)
                WIN = PATTERN
                RLS = RAILS
                OSC = NSC
                TYPE = ''
                .GT(DELTA,'0') /S(AGN)
                TYPE = ' INVERTED'
AGN             .NUM(GUESS)     /S(FINIS)
                M = DELTA * RAILS
                .LT(M,'0')      /S(AGN1)
                M = '+' M
AGN1            WRFLX('         ' M '           ' NSC)
                DELTA = '0' - DELTA
                .EQ(DELTA,'1') /S(MLP)
                N = RAILS       /(MLP2)
FINIS           WRFLX('')
                WRFLX('SOLUTION')
                WRFLX('SCORE ON ' RLS TYPE ' W STYLE RAILS IS ' OSC)
                WRFLX(WIN)
                /(TRY)
END             START
```

RAILFENCE PROBLEMS FROM THE CRYPTOGRAM

TOTIA SHHSI SESRL REEWE FTEBU OUMNN MOTNT FOAOA AC. (S-O 65)

ITEER DANSO EDXRE TNIRL EFNSE SDOOT BIRTY RRIRK CIEKE HDEAZ OENOH EHEOG LENEI NTTBU IP. (J-A 65)

DESDH DHSAN GEUHA TNNST AETYO OHNTY EIAPE LRIDN TGECM GTEDL OESOO HET. (M-J 65)

OEOWA RHRKS EPWHM KESTI ASPLO ITEHS LUTMO EERLI STWLA YMGEO E. (M-A 65)

ENELY MSVRI VAYOB BNYWI EONAH RTAIO WOLRM GHUGO BRSID LYINF DN. (J-F 65)

SITWS IIBHO VCUHE OEYVN ETAAA GUDEO IHRWO IGTOS DTLWN AIYNT LOULW IFADB AODFK DARFE S. (N-D 64)

RIOAN RISEW NGMTS DADAE HFMEH OEHTH KATTS LSTEL TRHTF ERYFH DEOLH VAKTS IROSI EIFTO TEFSH WHKTR S. (S-O 64)

VSDIH HLRAN ENITO SOTEL BCACO NESIN ERDLA EEBAI DHPSS HENAN CMBTW OTAWO Y. (J-A 64)

WUAAA IYRTB YLFBR AESIL FLAEH GATNA ALEOP OOIFE NFSO. (M-J 64)

ADHNR EDSAM NLOSI IHEHF SLARA BTEWE MDMFO TAIEA UENEH MLHSF NSLTA SWVRG TDNRO TOIEW AEHOO. (M-A 64)

SASEP AISNI CRPOB INGAF COEAH OCNNR NSOIS OIIAH RT. (M-A 63)

END OF PROBLEMS

[The above-listed problems where included as an appendage of the railfence program to facilitate testing. A more realistic cryptanalysis program would allow an analyst to type in a specific problem following the WHICH PROBLEM query of the program, rather than call a preloaded problem out of the program.]

This empty page was substituted for a blank page in the original document.

APPENDIX C

DETAILED DESCRIPTION OF OCAL SYNTAX

This appendix describes current specifications of the on-line cryptanalytic aid language. OCAL is intended to be a problem-oriented computer programming language, designed to make the solution of cryptograms easier by providing a cryptanalyst with computer aid. The ideas incorporated in OCAL have been taken from many languages, such as MAD, PL/I, SNOBOL, LISP, and SLIP. However, OCAL was not intended to have the full generality of a language such as PL/I. Instead, OCAL was originally specified for easy implementation on a computer such as the Digital Equipment Corporation PDP-6. As the design continued, some compromises were made to provide more features in the language, so that some of the specifications may change when the language is finally implemented on a computer.

C.1 SYNTAX NOTATION

In this appendix, meta-variables will be typed in small letters without intervening blanks, as the following:

identifier
label
boolean-expr

Capital letters indicate words that are part of the language, such as:

PROCEDURE
DO
STRING
BEGIN

The meta-symbol ... is used to indicate that an arbitrary number of the preceding meta-symbol can follow. All other punctuation marks such as . , : ; must appear as indicated. Optional portions of definitions will be set off using pairs of slashes [/]. For example,

LABEL name1/,name2,.../

means that the declaration LABEL is followed by at least one name and optionally, an arbitrary number of names separated by commas.

C.2 BASIC PROGRAM ELEMENTS

C.2.1 Character Set

The basic character set for OCAL is the revised ASCII character set. This character set is used for both language and data.

C.2.2 Identifiers

An identifier is a string of 29 or fewer alphanumeric characters; the initial character must be alphabetic. Identifiers are used for variable names, array names, statement labels, procedure names, and keywords.

C.2.3 Use of Blanks

Identifiers and constants (except string constants) may not contain blanks. Identifiers and/or constants may not be immediately adjacent. They must be separated by an operator, equal sign, paren, colon, semi-colon, period, or blank. All format effecters, such as horizontal tab, vertical tab, and line feed are treated as blanks, and multiple blanks are treated as one blank.

C.2.4 Comments

If the first character at the beginning of a line (i.e., after a Carriage-Return Line-Feed [CRLF] combination) is a star [*] then the entire line up to the next statement terminater (i.e., semi-colon or CRLF) is treated as a comment and is ignored in OCAL.

C.2.5 Statements

A statement is any single statement found in the language and is terminated by a semi-colon or a CRLF. Sometimes a statement can contain another statement as a sub-piece. (For example, see the IF statement). If a complete statement does not fit on one line, it may be continued on the next line by making the first character on the next line a period [.]. In this case, both the CRLF and the period are ignored by OCAL. This is true even within string constants.

C.2.6 Blocks

A block is a group of statements enclosed between the statements BEGIN and END. BEGIN and END act as statement parentheses and define a block. Blocks may be nested to any depth. A block may appear anywhere in the language a statement can appear, except that a block cannot appear in place of a declaration or PROCEDURE statement.

C.2.7 Statement Labels

Statements may be labeled to permit reference to them. A statement label has the form,

id:/id:.../ statement

where "id"s are identifiers. In this case, the identifiers are called statement labels and may be used interchangeably to refer to the labeled statement. Labels before procedures are special cases and are called procedure names (see Section C.7.1, PROCEDURE Statement). Only one label may appear before a PROCEDURE statement.

Statement labels appearing before declarations in OCAL are ignored.

C.3 BASIC DATA TYPES

C.3.1 Logic

A four-value logic is used in OCAL. The values and their meanings are:

T! - true
F! - false
N! - neutral or neither
U! - undefined

The logic values are ranked from lowest to highest, with N! lowest, then F!, T!, and U! highest. The result of logic constants combined under the operation .A. [AND] produces the lowest of the operands. Similarly, the operator .V. [inclusive OR] produces the highest of the operands. The operator .N. [NOT] inverts T! with F!, and N! with U!. The operator .X. [exclusive OR] behaves like .V. [inclusive OR] unless both operands are the same, in which case the result is the .N. [NOT] of the first operand.

C.3.2 Integer

An integer is an optionally-signed string of decimal digits, or an optionally-signed string of octal digits, followed by the letter K. For an octal integer, the K may be followed by an octal exponent given as a one- or two-digit decimal integer. The maximum size of an integer depends upon the particular OCAL implementation. On the PDP-6, up to ten decimal digits or twelve octal digits are permitted.

C.3.3 Real

A real number is an optionally-signed string of decimal digits including a decimal point [period]. In addition, a real number may have an exponent, indicated by the letter E, followed by an optionally signed one- or two-digit, decimal-integer exponent. The

maximum precision of real numbers is dependent on the particular implementation of OCAL. On the PDP-6, the exponent magnitude must be less than 10-to-the-38th power and the precision is limited to eight decimal digits.

C.3.4 Character

A character is a two- to eight-bit quantity representing an element of the ASCII character set mapped by an associated alphabet (see Section C.3.7). Characters are indicated in the language by a double quote mark ["] followed by one ASCII character or by a number sign [#] followed by exactly three octal digits. Characters may be mapped by alphabets from the ASCII character set to a subset of ASCII and back again.

For example, the ASCII character A may be represented by either of the following:

```
"A
#201
```

C.3.5 String

A string is an arbitrarily long sequence of ASCII characters delimited by single quote marks ['']. A string may contain any combination of ASCII characters. The characters single quote ['], double quote ["], and number sign [#] have special meaning when denoting a string in OCAL. Single quotes delimit the string, which means that one double quote mark is ignored and the character immediately following it is inserted in the string, no matter what that character may be. The double quote mark is used as a "quote" character, so that a single quote may be inserted in the string using the double quote mark. Since not all eight-bit ASCII characters can be generated from a normal teletypewriter keyboard, a special quote character, the number sign [#], is used to insert untypable characters in a string. A number sign must be followed by three octal digits, from 000 to 377, inclusive. This octal number represents the desired ASCII character.

Note that the carriage return and line feed characters may appear in a string. If a desired string will not fit on one line, the statement continuation convention may be used, in which case neither the CRLF nor the following period will appear in the string.

For example, the following all represent the same ASCII string in OCAL:

```
'ABC'
'A"B"C'
'#201B#203'
```

C.3.6 Reader

A reader is a special data type which may be associated with a given string. Using special reader functions, a reader may be moved up and down the string. A reader can also read characters from a string and write characters into a string (see Section C.9, Reader Functions). The reader was introduced into OCAL as a flexible way of transforming character strings into characters, and vice versa.

C.3.7 Alphabet

An alphabet specifies a mapping from the ASCII character set into ASCII. The idea was introduced into OCAL to add efficiency when dealing with characters as subscripts for compound data structures and arrays. Alphabets also allow core storage to be used more efficiently when storing character strings. In addition, alphabets can be used to exploit certain mathematical relationships often found between the characters of a particular cryptogram or cryptographic system. The alphabet declaration has two parts: the name, and the defining string given in OCAL string notation. In addition to the characters in the defining string, each alphabet includes two extra characters in the domain, standing for the logic values N! and U!. These are included to give OCAL the ability to indicate certain logical decisions within a string. However, the character corresponding to N! and U! are not included in the cardinality of the alphabet.

The declaration of an alphabet defines two objects within OCAL. First, a mapping function is called like an OCAL procedure which converts an ASCII string or character into a string or character in the given alphabet. Under this mapping, any character appearing in the domain (ASCII), but not in the range, is mapped into the null character (i.e., ignored). Second, the declaration permits the alphabet name to be used as a global integer variable whose magnitude is equal to the cardinality of the the defined alphabet.

An alphabet can also specify the mapping of many characters in the domain into one character in the range. This is accomplished by observing the following conventions in the defining string. All characters enclosed within parentheses in the defining string are mapped into the same character as the first character after the open parenthesis. If either of the literal characters open parenthesis "(" or close parenthesis ")" is desired in the range, it must be preceded by a double quote mark in the defining string.
(NOTE: a double quote mark is introduced into an OCAL string using the form "".)

For example, the following will declare a five-letter alphabet called A5, consisting of the characters A B C (and). In addition,

the ASCII characters D and E will be mapped into the character C.

```
ALPHABET A5('AB(CDE)""("")')
```

Using the alphabet A5, the ASCII string 'ABCDEF(ABZ)' will be mapped into the string 'ABCCC(AB)',

C.3.8 Type Transfer Procedures

The following procedures are available to transform quantities from one basic type to another. They are:

```
CHARACTER.(q)
```

where "q" is a logic, integer, or real quantity and the result is a character in the ASCII alphabet;

```
STRING.(q)
```

where "q" is a logic, character, integer, or real quantity and the result is a string in the ASCII alphabet;

```
LOGIC.(q)
```

where "q" is a character, integer or real quantity;

```
INTEGER.(q)
```

where "q" is a logic, character, ASCII string of digits, or real quantity; and

```
REAL.(q)
```

where "q" is a logic, character, ASCII string of digits in REAL form, or integer quantity.

The procedure

```
ASCII.(s)
```

will transform the string "s" in any alphabet to an ASCII string.

C.4 BASIC DECLARATIONS

In an OCAL procedure, each variable must be declared before it is used. The following forms are used to declare variables in an OCAL procedure:

```
LOGIC id/,id,id.../
INTEGER id/,id,id.../
REAL id/,id,id.../
CHARACTER id/,id,id.../
STRING id/,id,id.../
READER id/,id,id.../
ALPHABET id(st)
LABEL id/,id,id.../
EXTERNAL id/,id,id.../
GLOBAL id/,id,id.../
```

where "id" is an identifier and "st" is an OCAL string. The LABEL declaration means that the variable stands for a statement label.

The GLOBAL declaration means that the variable is to be made available to all OCAL procedures and is always defined. The EXTERNAL declaration means that the variable is a GLOBAL variable defined by some other OCAL procedure. The variables mentioned in a GLOBAL or EXTERNAL declaration must also appear within one of the type declarations. Variables not mentioned in a GLOBAL or EXTERNAL declaration are defined only within the procedure or block which contains the declaration.

C.5 COMPOUND DATA STRUCTURES

The compound data structures in OCAL are taken from the data structures found in the programming language PL/I. To avoid repetition of material, the following sections in Chapter 2 of the PL/I manual (IBM Form C28-6571-0) should be implemented in OCAL:

DATA AGGREGATES - page 43
- ARRAYS - page 44
- STRUCTURES - page 44
- ARRAYS OF STRUCTURES - page 44

NAMING - page 45
- SIMPLE NAMES - page 45
- SUBSCRIPTED NAMES - page 45
- QUALIFIED NAMES - page 46
- SUBSCRIPTED QUALIFIED NAMES - page 46

The only restriction on the data structures in OCAL is that blanks are not permitted within qualified names. In implementing these data structures in OCAL, it should be noted that each element of a compound data structure must be previously declared to be one of the basic data types, or must be a previously declared compound data structure. The recursive definition of a compound data structure is expressly prohibited in OCAL.

C.6 EXPRESSIONS

C.6.1 Arithmetic Expressions

The following infix operators are available for arithmetic expressions in OCAL:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

Arithmetic is performed on character, integer, and real data; the data types being ranked with character lowest, integer next, and real highest. The operands of any operator are converted to the

type of the highest operand, and the result is of that type unless one of the operands was a character. In that case, the result of the arithmetic expression is of character type and is taken modulo the size of the alphabet corresponding to the first character encountered. If this action is not desired, the following "dotted" operator set may be used:

.+. addition
.-. subtraction
.*. multiplication
./. division
.R. remainder

The "dotted" operators perform only the necessary type matching and indicated arithmetic.

C.6.2 Relational Expressions

Relational expressions return logic values and are used in making comparisons between various quantities. The relational operators are:

.G. greater than
.GE. greater than or equal
.L. less than
.LE. less than or equal
.E. equal
.NE. not equal

The operands may be of logic, character, integer, or real type. As in arithmetic expressions, type conversion takes place between character, integer, and real data types. However, if one operand is of logic type, then they both must be of logic type or the result will be U! [undefined]. Normally, the result of a relational expression is T! [true] if the relation holds and F! [false] if it does not.

C.6.3 Logic Expressions

The logical operators available in OCAL are:

.A. and
.V. inclusive or
.X. exclusive or
.N. not

The operands of a logical operator may be of logic type (ranked lowest), character, or integer (ranked highest). The result is of the same type as the highest operand and is the bit-wise combination of the operands according to the operator, unless both operands are of logic type. In this case, the truth tables indicated in Section C.3.1 are used.

C.7 STATEMENTS

C.7.1 PROCEDURE

The PROCEDURE statement marks the beginning of an OCAL function or procedure. It gives both the procedure name and the list of parameters the procedure is to receive. OCAL procedures may be recursively defined without any special declaration. The parameter list for a procedure may specify either a fixed or variable number of parameters. The form of the PROCEDURE statement for a fixed number of parameters is

```
id: PROCEDURE/(name1,name2,...)/
```

where "id" is the identifier giving the procedure name and the optional parameter list is enclosed in parentheses. Names in the parameter list give dummy names for arguments used by the procedure. Each dummy name must appear in a type declaration statement in the procedure.

For a variable number of parameters, the PROCEDURE statement has the form

```
id: PROCEDURE (/f,f,.../(v,v,...)/,f,f.../)
```

where "id" is the procedure name and the "f"s indicate optional parameters that are always present in the procedure call. The "v"s in parentheses indicate a set of parameters which may be repeated zero or more times in any procedure call. Again, all the dummy parameter names must appear in type declaration statements for the procedure. At each activation of the procedure, the global integer variable NUMBEROFPSETS will contain the number of parameter sets in this procedure call. Individual members of a parameter set may be referenced by the convention

```
parn[n]/(subs)/
```

where "parn" is the dummy name in the procedure parameter list, [n] is an integer or integer variable referring to a particular parameter set, and the optional (subs) is any subscripting expression associated with the parameter. Note that it does not make sense for the value of n to exceed the value of the integer variable NUMBEROFPSETS.

An OCAL procedure is terminated by an END statement (see next section). If control reaches an END statement for a procedure, it is equivalent to executing a RETURN statement with no return expression specified.

C.7.2 BEGIN AND END

The BEGIN statement or block marks the beginning of a compound statement which may appear any place a single statement can appear (except for a PROCEDURE statement or declaration). In addition, a

compound statement may start with type declaration statements, declaring local variables defined only within that compound statement or block. Variables used but not declared within a block are assumed to be declared in the procedure or in a block which encloses this one.

The statement

END /statement-label/

is used to terminate both a block and a procedure. The optional statement label, if present, must match the label on the corresponding BEGIN or PROCEDURE statement.

C.7.3 Assignment

The = sign is used to denote assignment in OCAL. This form gives

v1/,v2,v3.../ = e1/,e2,e3.../

where the "v"s are either variables which may be subscripted, or certain reader functions, and the "e"s are any OCAL expressions. If more than one variable or expression occurs, the assignments are made in pairs, e1 assigned to v1, e2 assigned to v2, etc. If there are more expressions than variables, the excess expressions are evaluated but the values are ignored. If there are more variables than expressions, the last expression value is assigned to the remaining variables.

Automatic type conversion is done within the following groups of data types:

character-integer-real
logic-character-integer

Assignments made to a character variable are made as stated, if the expression is of character type. Otherwise, the expression is taken modulo the size of the alphabet (if any) associated with the character.

C.7.4 PROCEDURE calls

Procedures are called with

procedurename./(p1,p2,...)/

This uses the MAD convention of following the procedure name with a period to differentiate it from a subscripted variable. The "p"s are optional parameters which, if present, are enclosed in parentheses. However, a statement may consist of only a procedure call, in which case any value returned by the procedure is ignored.

C.7.5 Iteration

The iteration statement DO allows a statement or block to be repeated zero or more times until some logical condition is met.

The DO statement takes the following forms:

DO UNTIL logicexpr, statement
DO WHILE logicexpr, statement
DO NEITHER logicexpr, statement

The UNTIL form repeats the statement until the logical expression logicexpr is not F! [false]. The WHILE form repeats while logicexpr is T! [true]. The NEITHER form repeats statement while logicexpr is N! [neither or neutral].

C.7.6 Conditional

The conditional statement takes the form

IF logicexpr, statement

If the logical expression "logicexpr" is T! [true], the statement is executed. Otherwise, the statement is skipped.

C.7.7 GO TO

The GO TO statement has the form

GO TO label

where label is a statement label or variable of LABEL type.

C.7.8 VALUE

The value returned by on OCAL procedure may be indicated by the statement

VALUE expr

where expr is any expression.

C.7.9 RETURN

A particular activation of an OCAL procedure, is terminated by executing the END statement associated with the procedure or by executing the statement

RETURN /expr/

The value returned by the procedure is the value of the optional expression "expr". If expr is not present, the value is taken from the last VALUE statement executed in the procedure. If expr is not present and no VALUE statement has been executed, the procedure returns a null value.

C.7.10 ERROR

A particular OCAL procedure may be terminated by the statement

ERROR /string/

Executing this statement causes control to return to the last ON ERROR statement executed (see ON statement). The value of the optional string associated with the last error statement is found as the value of the global string variable ERRORSTRING.

C.7.11 ON

The ON statement (an idea taken from PL/I) allows a programmer to retain control in spite of certain interrupts which might cause the OCAS job to terminate. The form of the ON statement is

ON condition, statement

where the "statement" (usually compound) is executed when the interrupt corresponding to "condition" is found. The interrupt conditions which the programmer can intercept with the ON statement are:

ERROR - error return from an OCAL procedure
CLOCKTICK - every time the system clock ticks
PDLOVERFLOW - overflow of push-down list
STORAGEFULL - no free storage left
DISBUFFERFULL - overflow of display buffer
DISPLAYSTOP - the display has executed a stop instruction
STORAGEUSED - allotted storage has been used (see Storage Allocation, Section C.10)
KEYSTROKE - one character has entered the on-line teletype buffer

If appropriate, the programmer can return control from the interrupt to the statement OCAL was executing when the interrupt occured by executing the statement

DISMISS

This permits OCAL to resume processing the previous calculation after some interrupt processing has been done.

The effect of an ON statement may be canceled by leaving the procedure in which the ON was executed, or by the statement

REVERT condition

which causes any interrupts corresponding to condition to be handled by an ON statement executed in a higher procedure.

The system may be requested to handle interrupts by the statement

SYSTEM condition

This instructs the system to do normal processing (if any) of any interrupt corresponding to this condition. The effect of a SYSTEM statement is canceled by leaving the procedure in which it was executed, or by executing a REVERT or ON statement specifying the same condition.

An interrupt on a particular condition may be simulated by the program by executing the statement

INTERRUPT condition

This has the same effect on OCAL as if the interrupt corresponding to condition had happened when the INTERRUPT statement was executed.

Once an interrupt corresponding to a certain condition has happened, further interrupts for the same condition are inhibited until a DISMISS statement has been executed or until an ON, REVERT, or SYSTEM statement specifing the same condition is executed.

C.7.12 SNOBOL Pattern Matching

The pattern-matching statement in OCAL is taken directly from the SNOBOL string-processing language. The basic forms of the SNOBOL statement are:

```
input /pe pe .../
input = st st /st .../
input pe pe ... = /st st .../
```

where "input" is a string or string variable, "pe"s are pattern elements (defined later), and "st"s are strings or string variables. The SNOBOL statement works in this manner: the input string is scanned from left to right for a match against the pattern elements in the given order. If a match is found and the = sign is present, matched pattern elements are replaced by the concatenation of strings "st" (if any).

Pattern elements may be string constants, string variables or arbitrary strings found in the input string itself. Arbitrary strings are denoted by string variables bracketed by stars.

For example: *A1*

HOLD

Arbitrary strings match any substring in the string input, including the null string. Arbitrary strings may be subject to a number of conditions. An arbitrary string designated

AA/3

will match a substring exactly three characters long. The general form of a fixed-length arbitrary string is

name/n

where "name" is a string variable and "n" is an integer or integer variable. An arbitrary string may be subject to the condition of containing a matching number of left and right parentheses. This condition is designated by

(name)

where "name" is a string variable.

An arbitrary string may be subject to a condition specified by a general logical procedure by using the form

name/proc.(arg1,arg2,...)

where "name" again is a string variable, "proc" is a logical procedure, and the "args" are any procedure arguments. The "args"

may specifically contain string variables which are substrings matched earlier in the SNOBOL pattern-matching statement. The logic procedure should return the value T! [true] if the proposed contents of name are satisfactory, N! [neither] if the proposed contents of name are not satisfactory because the string is too short, and the value F! [false] if the proposed contents of name are unsatisfactory for any other reason. If the logic procedure returns the value U! [undefined], the SNOBOL pattern scanner will take an ERROR return with the input string as the ERROR string.

After the pattern match is complete, the arbitrary string-variable names contain copies of the strings they matched in the input. These names may be mentioned in the concatenation section of the SNOBOL statement or in any other statement following the pattern-matching statement. Note also that string-variable pattern elements may have the same name as arbitrary pattern elements matched earlier in the pattern-matching statement. This makes it possible to search the input string for non-overlapping repeats of an arbitrary pattern element.

If the SNOBOL pattern match succeeds, the global logic variable SCANFLAG is set to T! [true]. Failure to find a match causes SCANFLAG to be set F! [false]. This condition can be tested by the IF or DO statements.

C.8 INPUT/OUTPUT PROCEDURES

Input/output procedures in OCAL will initially be limited to handling strings. Since the OCAL character set (ASCII) is quite general, strings can be converted to any other data type in OCAL. Conversely, output material can be converted to ASCII strings in OCAL. Two basic procedures are furnished with OCAL. They are:

READ.(file/,termin/)

WRITE.(file,string)

The argument file is either 'PTR', 'PNCH', 'TTY' or 'name1 name2' specifing photoelectric tape reader, paper tape punch, on-line teletype, or file names on backup storage (DECtape on the PDP-6). Only one file from backup storage may be open for reading and one file ophn for writing at a time. If the optional second argument "termin" is present in the READ call, the READ procedure returns as value the ASCII string of all characters up to and including the first match of the string termin. If termin is not present, the value of the READ procedure is all characters then in the input buffer. An end-of-file on backup storage is signaled by having the last character be ASCII character EOT.

The second argument of the WRITE procedure is the output string.

A file on backup storage may be closed by using the call

CLOSE.(file)

where "file" is a string 'name1 name2' as described above.

Examples:

INP = READ.('TTY','#215#212')

will read one line from the on-line teletype, up to and including the Carriage-Return (#215) Line-Feed (#212). The resulting string will be placed in the string variable INP;

WRITE.('PNCH',OUT)

will punch the contents of the string OUT on the paper tape punch;

IN = READ.('ALPHA DICT',' ')

will read from backup storage file ALPHA DICT the first string up to and including a space.

C.9 READER FUNCTIONS

Special functions for using the READER data type are available in OCAL. The general form of these functions is

$fn/fnfn.../.(readv)

where the "fn"s are elementary reader functions and "readv" is a variable of reader type. The elementary reader functions are:

C - Write one Character into a string if this appears on the left side of an assignment statement, otherwise read one character out of a string.

V - Set the reader position to the integer Value if this appears on the left side of an assignment statement. Otherwise return the integer value of the current reader position in characters from the head of the string.

I - Increment the reader position which moves the reader one character position forward on the string. If an attempt is made to pass the end of the string, the global logic variable ENDSTRING is set to T! [true]. Otherwise, the ENDSTRING is set to F! [false]. If the "I" is on the left side of an assignment statement and an attempt is made to pass the end of the string, the string is extended one character position and the global logic variable EXTENDSTRING is set to T! [true]. In any other case EXTENDSTRING is set to F! [false], and attempts to pass the end of the string leave the reader position unchanged and set the ENDSTRING variable.

D - Decrement the reader position which moves the reader one character position towards the beginning of the string. Any attempt to pass the beginning of the string will leave the reader position unchanged and the global logic variable BEGINSTRING set to T! [true]. If no attempt is made to pass the beginning of the string, BEGINSTRING is set to F! [false].

RI - Rotary Increment. This behaves like I [increment], except that passing the end of a string will position the reader at the beginning of the string.

RD - Rotary Decrement. This behaves like D [decrement], except that attempts to pass the beginning of a string will position the reader at the end of the string. No global variables are altered by RI and RD.

M - Mark. This notes the current position of the reader on the string for future reference.

P - Position. Return the reader to the position set by the last M [mark].

N - Initialize. Return the reader to the beginning of the string.

A reader may be attached to a given string by calling the ATTACH procedure with

```
ATTACH.(rdr,st)
```

where "rdr" is a variable of the READER type and "st" is any non-null string.

Example: (The following declarations hold throughout this example: R is a READER variable, S is a STRING variable, C and D are CHARACTER variables, and I is an INTEGER variable. The initial contents of S are 'LMNOPQ'.)

```
ATTACH.(R,S)
[attach reader R to string S]
C = $C.(R)
[set C equal to the character L]
D = $IC.(R)
[set D equal to the character M]
I = $VM.(R)
[set I equal to 2 and remember the value as a mark]
```

```
$V.(R) = 4
```

[position the reader over the character O]

```
$IC.(R) = D
```

[replace the character P with the character M]

```
$II.(R)
```

[this will produce no value but will set the global logic variable ENDSTRING to T![true]. The reader will be left positioned over the character R]

```
$IC.(R) = C
```

[set the global variable EXTENDSTRING to T! [true] and will append the character L to the end of the string]

```
$P.(R)
```

[return the reader to the mark. The reader will be positioned over the first M on the string]

```
$N.(R)
```

[return the reader to the head of the string]

As a result of previous reader functions, the string S will now contain 'LMNOMQL'.

C.10 RESOURCE ALLOCATION

Two resource allocation statements are available in OCAL. The statement

```
ALLOT PUSHDOWNLIST n
```

will allot "n" registers to the system push-down list where n is an integer or integer variable. The push-down list space allotment may be changed at any time, but an insufficient push-down list will cause a system interrupt.

The statement

```
LIMIT STORAGE n
```

will cause a system interrupt after n words have been used from free storage. The number of words of storage used since the beginning of the current OCAS job is found in the global integer variable STORAGEUSED.

Push-down overflow or storage-limit interrupt may be handled in OCAL by using the ON statement. These features allow the OCAL program to limit large searches or catch certain procedures that are in an infinite loop.

This empty page was substituted for a blank page in the original document.

APPENDIX D

ON-LINE CRYPTANALYTIC DISPLAY GENERATOR (OCDIS)

The following procedures will be available to generate CRT displays in OCAS. The inital implementation of OCDIS will be for the DEC Type 340 display attached to the Project MAC PDP-6.

D.1 PROCEDURES

The display is organized about a display format which is the argument to several display procedures. Only one format may be on the CRT at one time. Different formats may be thought of as different pages which may be displayed in any order under the control of an OCAL program. The basic display procedure is

COMPILEDIS.(frm/,q,q,q.../)

where "frm" is a string or string variable giving the display format and the optional "q"s are the variables or constants which are to be displayed in the given format.

Individual items within an already-compiled format may be named and named items may be changed using the procedure

CHANGEDIS.(frm,name,item)

where "frm" is the format, "name" is a string or string variable giving the name of the item in the format, and "item" is the new value of the quantity to be displayed. The advantage of this procedure is that individual display items in a large format may be changed without recompiling the entire display.

The display is started by the procedure

STARTDIS.(frm)

where "frm" is the format. In addition to the requested format, each STARTDIS will cause a log display to appear in the upper left hand corner of the screen. The log gives the current date, time, frame number, and a short title for the display. The log information is useful in identifying still photographs taken of the display and is maintained by the system without being a burden to the programmer. The log information for a particular console session may be initialized using the procedure call

LOG.(date,time,frame,title)

where "date" is an integer giving the current Julian day, "time" is an integer giving the current time in 60ths of seconds after midnight, "frame" is the inital frame number, and "title" is a short string used to title the display. A negative number in the date, time, or frame positions will leave those constants unchanged. The frame number is incremented by one every time a new format is displayed. The system will automatically update the date, time, and frame number once they are initialized.

The display may be turned off using the procedure call

STOPDIS.

Room for the display buffer may be dynamically allocated by calling the procedure

BUFFERDIS.(n)

where "n" is an integer variable or constant giving the size of the display buffer in words of core storage. If the display buffer is too small for a particular display, the buffer will overflow. This condition may be detected in an OCAL program with the ON statement using the DISBUFFERFULL condition.

All displays are maintained in program interrupt mode, so that calculations may continue even when a display is visible.

D.2 FORMATS

A format is an ASCII string in the form

'x y item item ...

where x and y are octal integers giving the absolute reference point in screen co-ordinates for the rest of the format. Each item is a list of display descriptors; the entire list for any one item being enclosed in parentheses. The display descriptors for a particular item may be in any order and only those descriptors relevant to the item being displayed need be included in the list. Certain display descriptors, such as SIZE, INTENSITY, and RELOC effect each item in the display. If they are not specified for a particular display item, the previous item's value is used for SIZE and INTENSITY, and the RELOC is taken from wherever the last item finished.

D.3 DISPLAY DESCRIPTORS

Each display descriptor is enclosed in parentheses. It consists of a descriptor type followed by modifiers or values separated by spaces. The display descriptors are:

(TYPE t) - gives the basic type of data quantity to be displayed as this item. Permitted types are STRING, CHARACTER, INTEGER, REAL, and LOGIC.

(RELOC x y) - the octal integers x and y give the starting location of the display item in screen co-ordinates relative to the format reference location. If RELOC is not specified, the item will be displayed starting wherever the last item stopped.

(NAME nm) - gives the external name of this display item. Named items may be changed without recompiling the entire display format by using the CHANGEDIS procedure.

(SIZE n) - where n is a decimal integer from 1 to 4 giving the character size or dot separation to be used. (See the Type 340 section of the PDP-6 manual.)

(INTENSITY n) - where n is a decimal integer from 1 to 8 giving the relative intensity of the displayed item.

(CASE c) - where c is UPPER or LOWER. This is used to determine the case of an alphabetic character or string display

(SPACE n) - where n is NO or a decimal integer. This descriptor is for string displays. NO causes spaces in the string to be suppressed. An integer will cause a space to be inserted after every nth letter (5 is a typical value).

(WIDTH n) - where n is a decimal integer. This descriptor sets the width in characters for a string display. If the string to be displayed is longer than n characters, the string is broken into lines of length n. The space between successive lines is normally one vertical character space, but this may be increased to n character spaces using the VSPACE descriptor.

(DEPRESS n) - where n is a decimal integer. This descriptor, used in string displays, declares that the string should begin n vertical character spaces below the position specified by RELOC.

(VSPACE n) - where n is a decimal integer giving the number of vertical character spaces between successive lines of a string display.

(BASE) - declares this item to be a control string which is not displayed. The control string is used as a reference for the variable spacing descriptor BELL.

(BELL n) - where n is a decimal integer number of characters. It is used to prevent words in a string from being broken between successive lines in the display. BELL causes the display line to end at the first space after n characters relative to the BASE reference string. If no BASE string is specified, the string being displayed will be used as the reference string.

(OFFSET) - declares that the next parameter in the argument list is an integer offset to be applied to the current string. This descriptor is useful when displaying cryptographic slides.

(CONSTANT ...#000) - every character after the space following CONSTANT up to the special terminating character #000 is taken as a constant string to be displayed. CONSTANT's need no argument in the corresponding position in the COMPILEDIS call.

(ARRAY n1:m1 /n2:m2/) - where the arguments are array subscript ranges in the OCAL format. This descriptor is used to declare that the display argument is a one- or two-dimensional ARRAY. Only arrays of type CHARACTER, INTEGER, or REAL may be displayed.

(BARGRAPH n:m) - where n and m are integers. This descriptor indicates that the display is a one-dimensional array that is to be displayed as a bar graph. Only INTEGER or REAL arrays may be displayed as a bar graph.

(SCALEFACTOR) - is used with the BARGRAPH descriptor. It indicates that the next item in the call is a real number which is to multiply each item in the bar graph display.

(LINE x y) - where x and y are octal integers. This causes a line to be drawn from the current relative location to the point x,y relative to the format reference point. The line may be solid or dotted, depending on the SIZE descriptor.

APPENDIX E

ON-LINE DEBUGGING AND CONTROL PROGRAM (ODBUG)

This appendix describes the features of the on-line debugging and control program for OCAS. The program makes use of an OCAL interpreter so that an OCAL statement may be executed by typing it on the console. In addition, the following features are included:

:var - causes the contents of the variable "var" to be printed out on the on-line console. After the printout the varible is "open", which means new contents may be inserted by typing them using OCAL conventions. A statement terminater "closes" the variable. If nothing is typed before the statement terminater, the contents of the variable remain unchanged.

/S - causes the current OCAL table of active symbols to be typed out giving both the symbol names and their types.

/D name - causes the entire current state of OCAS to be dumped on backup storage in a file called NAME SAVED.

/R name - restores the state of OCAS from the NAME SAVED file on backup storage.

/P pro - where pro is the name of an interpreted OCAL procedure. This permits ODBUG to insert a breakpoint in this procedure (see /B).

/B id - places a breakpoint at the statement label "id" in the currently-addressed OCAL procedure. Executing a breakpoint returns control to ODBUG. If "id" is not specified, any outstanding breakpoint is removed.

/C - allows OCAS to continue executing statements after the last breakpoint was executed.

/G id - starts the OCAL interpreter at the statement label "id" in the currently-addressed procedure (see /P).

BREAK - a single depression of the BREAK button will return control to ODBUG as if a breakpoint had been executed. The program may be restarted using the /C command.

This empty page was substituted for a blank page in the original document.

APPENDIX F

AN EXAMPLE IN OCAL - FINDING THE PERIOD OF A PERIODIC CIPHER

The following example is based on a method suggested by William G. Bryan in Cryptographic ABC's for finding the period of a cryptogram enciphered with a periodic cipher (e.g., Vigenere or Beaufort). The method consists of finding the distance in characters between each and every A in the cryptogram. The distances are then factored and tallies are made for each factor corresponding to a suspected period of the cryptogram. Usually a range of periods from 3 to 12 is tested. This procedure is repeated for each B, each C, etc., down to each Z. Next, the tallies corresponding to each period are summed and weighted by the period. The highest weight usually indicates the period of the cryptogram.

This method of finding the period is known as the "Kasiski" method after Major F. W. Kasiski, a German cryptanalyst, who published a paper on it in 1863 (see page 127 in GAINES). This method works because in a periodic cipher, the key must be repeated a number of times to produce a cryptogram and, as a result, many times the distance between two occurences of the same cipher text letter is a multiple of the key length which is the period.

EXAMPLE

```
*
*   KASISKI METHOD IN OCAL
*
PERIOD:  PROCEDURE (CRYPT,PER,N,M)
*
*  PARAMETERS ARE:
*    CRYPT - A STRING GIVING THE CRYPTOGRAM
*    PER - AN INTEGER VECTOR WITH SUBSCRIPT RANGE N TO M
*          THE WEIGHTED TALLIES ARE RETURNED IN THIS VECTOR
*    N - AN INTEGER GIVING THE LOWEST PERIOD TO BE TESTED
*    M - INTEGER GIVING HIGHEST PERIOD TO BE TESTED
*
*  DECLARATIONS NEXT
*
        STRING CRYPT
        CHARACTER C
        INTEGER PER,N,M
        INTEGER DIST,INDEX,ALPS,K,L1,L2
        INTEGER SEP,TR
        READER R
        DECLARE PER(*)
*
*  THE VECTOR PER IS DIMENSIONED IN THE CALLING PROCEDURE
*
        DECLARE DIST(LENGTH.(CRYPT)/5)
*
*  DECLARING THE LOCAL VECTOR DIST
*
```

```
         READER R
*
* THE ACTUAL PROCEDURE BEGINS HERE
*
         ATTACH.(R,CRYPT)
         ALPS = SIZE.(ALPHABET.(CRYPT))
         INDEX = 1
         DO WHILE INDEX .LE. ALPS, LOOP1: BEGIN
*
* ITERATE OVER THE SIZE OF THE ALPHABET
*
            C =$NC.(R)
*
* RETURN READER TO HEAD OF STRING AND READ FIRST CHARACTER
*
            K = 1
            DO UNTIL ENDSTRING, LOOP2: BEGIN
*
* READ THE STRING CRYPT CHARACTER BY CHARACTER
*
               IF C .E. INDEX, COND1: BEGIN
                  DIST(K) = $V.(R)
*
* RECORD DISTANCE FROM HEAD OF STRING
*
                  K = K + 1
                  END COND1
               C = $IC.(R)
*
* INCREMENT THE READER AND READ NEXT CHARACTER
*
               END LOOP2
            L1 = 1
            DO UNTIL L1 .E. K, LOOP3: BEGIN
*
* COMPUTE THE CHARACTER DISTANCE BETWEEN EACH OCCURENCE
*
               L2 = L1 + 1
               DO UNTIL L2 .G. K, LOOP4: BEGIN
                  SEP = DIST(L2) - DIST(L1)
                  TR = N
                  DO UNTIL TR .G. M, LOOP5: BEGIN
*
* TEST EACH PERIOD FROM N TO M FOR REMAINDER 0
*
                     IF (SEP .R. TR) .E. 0,
.                       PER(TR) = PER(TR) + 1
                     TR = TR + 1
                     END LOOP5
                  L2 = L2 + 1
                  END LOOP4
               L1 = L1 + 1
               END LOOP3
            INDEX = INDEX + 1
            END LOOP1
*
* NOW WEIGHT EACH ITEM IN PER BY THE RESPECTIVE PERIOD
*
         K = 1
         DO UNTIL K .G. M, LOOP6: BEGIN
            PER(K) = PER(K) + 1
            K = K + 1
            END LOOP6
*
* ALL DONE
*
         END PERIOD
```

BIBLIOGRAPHY

ARDEN, Bruce et al, The Michigan Algorithm Decoder (MAD), University of Michigan, November, 1963

BAZERIES, Commandant E., Les Chiffres Secrets Devoiles, Paris, 1901

BOBROW, Daniel G., "METEOR: a LISP Interpreter for String Transformations", The Programming Language LISP: Its Operation and Applications, Information International Inc., Cambridge, Massachusetts, 1964

BRYAN, William G., Cryptographic ABC's, American Cryptogram Association, 1960

DEC, DDT-6 Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, 1965

DEC, Programmed Data Processor-6 Handbook, Form F-65, Digital Equipment Corporation, Maynard, Massachusetts, 1965

EYRAUD, Charles, Precis de Cryptographie Moderne, 2nd edition, 1959

FARBER, David, et al, "SNOBOL, A String Manipulation Language", Journal of the Association of Computing Machinery, Vol. 11, No. 2 (January 1964), pp. 21-30

FRIEDMAN, William F., An Introduction to Methods for the Solution of Ciphers, Riverbank Laboratories Publication No 17, Geneva, Illinois, 1918

GAINES, Helen F., Cryptanalysis, Dover Publications, New York, 1956

GRISWOLD, Robert E. and POLANSKY, I. P., String Pattern-Matching in the Programming Language SNOBOL, Memorandum MM-63-3344-3, Bell Telephone Laboratories Inc., July, 1963

IBM Operating System/360 - PL/I: Language Specifications, Form C28-6571-0, 1965

McCARTHY, John, et al, LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, 1963

NAUR, Peter, et al, "Revised Report on the Algorithmic Language ALGOL 60", Communications of the Association for Computing Machinery Vol. 6, No. 1, (January 1963), pp 1-17

PRATT, Fletcher, Secret and Urgent, Blue Ribbon Books, New York, 1939

SACCO, General L., Manuale di Crittografia, 2nd edition, Rome, 1936

WEIZENBAUM, Joseph, "Symmetric List Processor (SLIP)", Communications of the Association of Computing Machinery, Vol. 6, No. 9 (September 1963), pp. 524-536

YARDLEY, Herbert O., The American Black Chamber, Blue Ribbon Books, New York, 1939

YNGVE, Victor H., et al, COMIT Programmers Reference Manual, MIT Press, Cambridge, Massachusetts, 1961

ZANOTTI, Mario, Crittographia: Le Scritture Segrete, Milan, 1928