*Interim Technical Report 2*

# MOL940:
# PRELIMINARY SPECIFICATION FOR AN ALGOL-LIKE MACHINE-ORIENTED LANGUAGE FOR THE SDS 940

*By:* R. E. HAY J. F. RULIFSON

*Prepared for:*

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
LANGLEY RESEARCH CENTER
MAIL STOP 126
LANGLEY STATION
LANGLEY, VIRGINIA 23365

CONTRACT NAS1-5904

March 1968

STANFORD RESEARCH INSTITUTE

MENLO PARK, CALIFORNIA

STANFORD RESEARCH INSTITUTE

MENLO PARK, CALIFORNIA

March 1968

*Interim Technical Report 2*

# MOL940: PRELIMINARY SPECIFICATION FOR AN ALGOL-LIKE MACHINE-ORIENTED LANGUAGE FOR THE SDS 940

*By:* R. E. HAY J. F. RULIFSON

*Prepared for:*

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
LANGLEY RESEARCH CENTER
MAIL STOP 126
LANGLEY STATION
LANGLEY, VIRGINIA 23365

CONTRACT NAS1-5904

*SRI Project 5890*

Study for the Development of Human
Intellect Augmentation Techniques

*Approved:* BONNAR COX, ACTING MANAGER
SYSTEMS ENGINEERING LABORATORY

T. H. MEISLING, EXECUTIVE DIRECTOR
INFORMATION SCIENCE AND ENGINEERING

Copy No. 81

ABSTRACT

This report is a reference manual for a programming language developed at Stanford Research Institute for the Scientific Data Systems 940 computer. The compiler is now fully operational; it is written in its own language, compiles itself, and is in daily use for development of our CRT-display service system.

The name MOL940 (or simply MOL), is an acronym for "Machine-Oriented Language." MOL is an ALGOL-like language with natural extensions for bit manipulation. The added syntax strongly reflects the internal design of the SDS 940, in accordance with the name MOL.

The introduction to this report includes a brief summary of other projects of the same nature which were known to the authors. There is also a discussion of the design criteria that shaped the MOL. The major topics are the comprehensibility of programs written in the language, the needs of system programmers working within a time-sharing system, and the effects on coding that result from using an on-line CRT.

A complete definition of the language is given, using an extended Backus Normal form; included are semantic explanations and examples, a sample program, and some examples of code produced by the MOL compiler.

TABLE OF CONTENTS

FOREWORD

Development of the MOL (Machine-Oriented Language) began in October 1966 under joint sponsorship of the Advanced Research Projects Agency, Defense Department, and NASA, Langley Research Center. Although completion took approximately one year, only six man months have been invested in the project. The Augmented Human Intellect (AHI) Program (ENGELBART1) is using the MOL as the base language for its software effort. The language and compiler have been explicitly designed to facilitate concurrent modification and development of AHI programming techniques.

This report has been prepared with the On-Line Text Manipulation System, and consequently it differes in a few respects from other technical reports. All paragraphs are hierarchically numbered; certain paragraphs bear "names," and references appear as an author's name, perhaps with a sequence number, enclosed in parentheses.

1 Original computer language development was guided by the already existing formalisms of the numerical analysts. The machine- independent evolutionary direction of the problem-oriented languages has enhanced their algorithmic and algebraic nature, but destroyed their usefulness as system program languages.

1a The concerns of system programmers such as efficiency, tight code, and bit manipulations require a different orientation. Machine independence and algebraic constructs are not discarded but are enhanced; additional features are included to permit succinct, explicit references to hardware functions necessary in systems programming on a display-oriented time-sharing computer.

2 Erwin Book (See BOOK1) of System Development Corporation supplied the original impetus for our new language with his Q-32 machine-oriented language (MOL). Niklaus Wirth simultaneously undertook a similar project while at Stanford University. His PL-360 (See WIRTH1) was designed as a precedence grammar (See WIRTH2) and used to implement a version of ALGOL on the IBM 360.

3 Our aim throughout the development of MOL 940 was to design a coordinate language-compiler pair that permits the expression of clear, concise algorithms and the production of efficient, tight code. With such a language, fewer bugs slip in during coding, programmers can say what they want in fewer words, and (with a little luck) one can pick up some of his year- old code and understand it.

3a Algorithmic clarity is mainly due to the structure implicit in the syntax of the language.

3a1 It is significant in this regard that labels have almost disappeared in existing MOL code. Instead the CASE and WHILE statements are the primary means of controlling program flow (See WIRTH3). The program is not interleaved with many GOTO statements transferring into and out of sections of code so that only the original programmer can remember all the ways a certain statement may be reached. Just the way MOL code appears on a page makes the algorithmic flow clear (See SCHORRE1).

3a2 The succinctness of infix notation rather than assembly language also adds clarity. It is often quite difficult to pick up a random page of machine code and recognize that a set of five lines doing very strange things are actually testing for a flag in a word, but it is very easy to recognize a line of MOL and "see" a test being made.

3b Our concern for the production of tight code led us to believe

that programmer and compiler must work together; the compiler alone cannot do the job.

3b1 While the programmer can do the job alone, it is usually too time consuming. The idiosyncrasies of the SDS 940 are reflected in the special constructs incorporated in the MOL which allow the programmer control of the code that is generated and the way in which registers are used.

3c We have also included rather general expressions and assignment statements in the MOL. At times the programmer has no need for tight code and should be able to use the MOL on a higher level, leaving all the worrying about final constructs to the compiler.

3d A unique consideration within the MOL design criteria is the accommodation of potential coordination between the structure of the language and a display-oriented time- sharing text editor.

3d1 With such a system, there may not exist hard copy and the programmer would be able to see no more than some twenty lines of his program at any one time.

3d2 Much can be done to ease the programmer's movements within the code to facilitate manipulation of logical chunks of code and to allow at least everything that can be done with cards and a listing.

3d3 We would like to give the programmer even more for we feel that our text structure conventions and the associated features of NLTS can be used for algorithm analysis; these techniques, coupled with the design of the language and of the compiler, provide greater power and facility for dealing with program design than more conventional methods such as flow charts. In ESD1 we presented some basic discussion in this direction (See ENGELBART5).

4 The MOL 940 compiler uses a META compiler parser and a general operator-operand stack for the code producing algorithm. Additions to the syntax take only minutes to implement. As a result we do not try to plan for all future constructs. Instead, our attitude of restraint means that syntax is added when the need arises and the style of the construct is well thought out.

# 1 Terminology

1a The syntax for the MOL language is written in the META II notation. This provides an easy means of expressing the syntax in a form that is readable by both man and machine, yet allows great ease and flexibility in modifying the constructs that describe the language.

1a1 The notation used for the META II syntax, as well as for the MOL language, is quite similar to the notation used in the ALGOL 60 report.

1a2 Terminal symbols are represented as strings of characters bounded by quotes. Nonterminal symbols take the form of an ALGOL identifier (i.e., a letter followed by a sequence of letters or digits).

1a2a Any terminal symbol consisting of a single character may be preceded by a single quote rather than enclosed in quotes to indicate that it is terminal.

1a3 Concatenation is designated by writing items consecutively. The items are separated by slashes to indicate alternation. Each syntax equation ends with a semicolon.

1a4 A special syntactic entity represented as ".empty" has been incorporated to indicate that a syntactic element is optional, and is usually used in conjunction with alternation.

1a5 Also, it is possible to "factor" part of a syntax equation; that is, parentheses can be used to group a sequence of items so as to treat it as a single item.

1a6 A special operator, m$n (where m and n are optional integers), is also used to designate "any number between m and n of occurrences of the following item." The default values of m and n are zero and infinity. This makes it possible to reduce the number of equations needed to obtain recursion on some item in the syntax. For example, the standard definition of identifier now becomes:

```
1a6a  identifier = letter $(digit / letter);.
```

1b The design philosophy for the MOL compiler was to follow the META II design, i.e., that of recursive recognizers. The reasons for this choice center around the following considerations:

1b1 Most of the people using and designing the MOL language and compiler have had direct experience writing recursive recognizer

compilers.

1b2 To design a precedence grammar and compiler means that the relationship between each character and all other characters has to be considered at each point, and an arbitrary construct cannot be added at will without possibly affecting the rest of the existing relations.

## 2 Basic Symbols and Syntactic Entities

### 2a General Vocabulary

#### 2a1 Terminal Vocabulary

2a1a A B C D E F G H I J K L M N O P Q R S T U V X Y Z 1 2 3 4 5 6 7 8 9 0 ( ) ← ' $ * : = - + ; ↑ , . / < > [ ]

2a1b AND BEGIN BUMP BY CALL CASE DECLARE DO DO-SINGLE ELSE END ENDP. ENTRY EXECUTE EXTERNAL FINISH FOR FROZEN GO GOTO IF INC NOT NULL OF OR POP PREFIX PROCEDURE PROC RETURN SET STEP THEN TO UNTIL VIRTUAL WHILE .A .E .V .LT .LE .EQ .NE .GE .GT .CB .NCB .AR .BR .XR .BRS .LSH .LCY .LRSH .RCY .RSH

#### 2a2 Nonterminal Vocabulary

2a2a <abxreg> <actual> <act1> <act2> <act3> <act4> <address> <arpas> <assign> <band> <bexp> <block> <bor> <bound> <builtin> <bump> <call> <case> <constant> <cvar> <decl> <declaration> <entry> <equ> <equ1> <exp> <ext> <exu> <factor> <for> <formal> <form1> <form2> <form3> <form4> <frozen> <frz1> <goto> <icon> <if> <immediate> <index> <indirect> <intersection> <item> <iterative> <labeled> <negation> <null> <parid> <prefix> <primary> <procedure> <product> <relation> <return> <simple> <statement> <sum> <union> <value> <variable> <varfun> <virtual> <while>

### 2b Primitives

2b1 Identifiers: An identifier is a symbol used to name a quantity (such as a procedure, a variable, or an array), as a label or formal parameter.

2b1a Syntax: id = letter $5(letter / digit);.

2b1b Semantics: An identifier (or more simply an id) is a string of letters and digits, with a maximum length of 6, the first of which must be a letter.

2b1b1 All identifiers that are local to a procedure must be

declared at the beginning of the procedure. Those variables not declared or used as labels are assumed to be virtual, i.e., defined in some other procedure. No distinction is made among array, procedure, and label uses of identifiers.

2b1c Examples of Identifiers:

2b1c1 I

2b1c2 CHAR

2b1c3 X21

2b1c4 I2JBY1

2b2 Numbers

2b2a Syntax: number = 1$8 digit ("b" / .empty) ;.

2b2b Semantics: A number is a string of digits, with a maximum length of eight characters, possibly terminated with a letter b. If the terminating character is a b, then the number is taken to be octal; otherwise it is taken to the base 10.

2b2c Examples of Numbers:

2b2c1 1

2b2c2 1024

2b2c3 77770000b

2b3 Strings

2b3a 8-bit character strings are the only strings recognized by the MOL compiler, and these can only occur in declarations.

3 Declarations: All declarations occur at the start of a procedure, as declarations are not allowed within a block. All variables declared in a procedure become local to that file (not just the procedure), and external to that file, if so declared. Variables can be preset, arrays declared, and virtual symbols specified.

3a Procedure: The procedure is the basic syntactic entity, in that one writes procedures, which are compiled, assembled, and loaded.

3a1 Syntax: procedure = parid ("pop" "(" .num "," .num "," .num ")" /.empty) ("procedure" /"proc") formal ";" $declar labeld $(";" labeld) "endp.";

3a1a parid = "(" id ")" ;

3a1b formal = "(" ( id / .empty) $2("," ( id / .empty)) ")" ;.

3a2 Semantics: The procedure declaration begins with an indentifier which serves as the name of the procedure. Optionally, one can declare a procedure to be a "POP" procedure so that it will be treated by the system as a user POP.

3a2a Following the word "procedure" one optionally indicates the parameters to this procedure. A maximum of 3 is allowed, to correspond to the A, B, and X registers, which are the only arguments passed when a call to a procedure is made. These parameters are indicated by placing them after the word "procedure," and enclosing them in parentheses.

3a2b After the procedure declaration comes a declaration of all the variables that are to be used in that procedure, their dimensions (if any) and their values if they are being preset.

3a2c The sequence of statements that constitutes the executable code of the procedure follows these declarations. In this, note that one cannot declare variables within blocks, and that variables can only be declared at the beginning of a procedure.

3a2d Finally, all procedures must end with an "endp".

3a3 Example of Procedure:

3a3a (get) procedure(x,i); declare x,i; return([x[i+1]]↑4) endp.

3b Declaration

3b1 Syntax: declaration = ( decl / ext / equ / virtue / frozen / prefix ) ";" ;.

3c Decl

3c1 Syntax: decl = "declare " ("external " /.empty) item $("," item);

3c1a item = .id (bound /.empty) (value /.empty) ;

3c1b bound = "[" (.id /.num) "]" ;

3c1c value = "="( "(" icon $("," icon) ")" / icon );

```
3c1d  icon = (.num /.id /.st8) ;.
```

3c2 Semantics: The basic declaration statement permits declaration of those variables which are to be allocated in the current procedure (and possibly made external to the current file, to indicate their dimensions (if arrays), and to specify the values to which they are to be preset (numbers, addresses of identifiers, or strings).

3c3 Examples:

```
3c3a  declare x,y,z[10];

3c3b  declare external m=10,n=m,st='end of file';

3c3c  declare sk[10]=(0,1,20,40);.
```

3d External

3d1 Syntax:

```
ext = "external " evar $("," evar);

3d1a  evar = .id ;.
```

3d2 Semantics: The external declaration generates "ext" records for the assembler--that is to say, those variables following the "external" are defined to be external to the current file, but they are not allocated any storage. In this last respect they differ from variables which are declared via the "declare external" statement. "External" is sometimes used to declare labels to be external.

3d3 Example

```
3d3a  external m,n,z;.
```

3e Equate

3e1 Syntax:

```
equ = "set " equ1 $("," equ1) ;

3e1a  equ1 = .id "=" (.id /.num) ;.
```

3e2 Semantics: The equate declaration generates "equ" records for the assembler--that is to say, those variables that are indicated are equated to the value given at assembly time. This is useful in generating conditional assemblies, and in setting the array bounds via a "set " identifier.

3e3 Example:

3e3a set m940=1,skmax=100;.

3f Virtual

3f1 Syntax: virtue = "virtual " cvar $("," cvar) ;

3f1a cvar = .id (bound /.empty);.

3f2 Semantics: If a variable is not declared in a file, then it is known as virtual. Via the "virtual" declaration, it is possible to tell the compiler which variables are expected to be virtual; appropriate checks can then be made, and when the cross-reference listing is generated, these variables will be marked "v" for virtual, instead of "u" for undefined.

3f3 Example:

3f3a virtual a,b,m[32b];.

3g Frozen

3g1 Syntax: frozen = "frozen " frz1 $("," frz1);

3g1a frz1 = .id;.

3g2 Semantics: The frozen declaration is used to tell the compiler that the following variables are local to this file, but that no storage should be allocated for the variables. This distinction is needed because the codes for local and virtual variables are different. Since the loader links undefined symbols together through the address field, it is not possible to have a complex address field (such as " lda m+1") for a virtual symbol. Thus for the compiler to generate the appropriate index register loads and the correct address field, it needs to know whether a variable is local or virtual. The frozen declaration is a way of making the compiler think that a variable is local when it is virtual. This is used in connection with the ARPAS "continue assembling", and "frozen symbol table" features.

3g3 Example:

3g3a frozen a,b,x;.

3h Prefix

3h1 Syntax: prefix = "prefix " "for " ("generated " "labels:" .st8 /"temporaries:" .st8) ;.

3h2 Semantics: By using a higher-level language, it is possible

to have the compiler generate labels and temporaries which, at the machine-language level, would otherwise have to be done by the user. However, the compiler is now generating labels and temporaries, using identifiers that are the same for each compilation. For debugging, and for generating reentrant code, it is useful to be able to specify different names. The "prefix" declaration permits the user to specify the names used for the generated labels and temporaries.

3h3 Examples:

3h3a prefix for generated labels: 'fmt';

3h3b prefix for temporaries: 'libet' ;.

4 Expressions: An expression is an entity which represents a numerical value (contained in one 24-bit word). This value is obtained by using the values of the identifiers and functions within the expression, and combining these values by means of the operators within the expression. Note that the symbols .ar, .br, and .xr are associated with the internal registers of the machine, and their values are the contents of the respective registers.

4a Exp

4a1 Syntax: exp = "if" bexp "then" bexp "else" exp / bexp ;.

4a2 Semantics: A general expression can be either a conditional expression, using the "if then else" type of construct, or it may be an expression resulting from the combination of arithmetic, Boolean, or relational operators.

4a3 Examples:

4a3a if x .le y then 1 else 2

4a3b x+y*z/(x+1)

4b Bexp

4b1 Syntax: bexp = union;.

4c Union

4c1 Syntax: union = intersection $("or" union);.

4c2 Semantics: The union makes it possible to combine expressions with the logical operator "or." The result of the "or" operator is true (i.e. not equal to zero) iff at least one of the

expressions is true.

4c3 Example:

4c3a x or y

4d Intersection

4d1 Syntax: intersection = negation $( "and" intersection);.

4d2 Semantics: The intersection makes it possible to combine expressions with the logical operator "and." If both expressions are true, then the result will be true.

4d3 Example:

4d3a x and y

4e Negation

4e1 Syntax: negation = "not" negation / relation;.

4e2 Semantics: This construct makes it possible to take the (logical) negation of the value of any expression.

4e3 Example:

4e3a not x

4f Relation

4f1 Syntax: relation = sum (".gt" sum /".ge" sum /".ne" sum /".eq" sum /".le" sum /".lt" sum /".cb" sum /".ncb" sum /.empty );

4f2 Semantics: The relational operators make it possible to construct logical statements which are true if the given arguments stand in the specified relation to one another. The operators are "greater than," "greater than or equal," "not equal," "less than or equal," "less than," "common bits," or "no common bits." The "common bits" operator yields a value of true iff both of its arguments have ones in any corresponding bit positions. The "no common bits" operator yields a value of true iff its arguments do not have ones in any corresponding bit positions.

4f3 Examples:

4f3a m .gt n

4f3b z .ne y

4f3c x .cb y

4g Sum

4g1 Syntax: sum = product $("+" product / "-" product);.

4g2 Semantics: The sum permits one to combine expressions with the arithmetic operators + and - . Note that all values are taken to be 24-bit integers.

4g3 Examples:

4g3a x

4g3b x + y

4g3c x - y + z

4h Product

4h1 Syntax: product = factor $("*" factor / "/" factor / "↑" factor);.

4h1a Syntax: factor = bor / "-" factor ;

4h2 Semantics: The product permits one to combine expressions with the arithmetic operators * (times), / (division), and ↑ (mod). The result of these operators is a 24-bit integer, and in the case of the division, the remainder is discarded. Mod operates similarly to division except that the quotient is discarded and the remainder is the result of the operation.

4h3 Examples:

4h3a x

4h3b x * y

4h3c x / y

4h3d x ↑ y

4i Bor

4i1 Syntax: bor = band $(".v" band / ".x" band);.

4i2 Semantics: The "bor" (standing for "bit or") makes it possible to obtain the bitwise "or" of two expressions. Both inclusive and exclusive "or" are allowed and are designated by .v

and .x respectively.

4i3 Examples:

4i3a x

4i3b x .v y

4i3c x .x y

4j Band

4j1 Syntax: band = primary $(".a" primary);.

4j2 Semantics: The "band" (standing for "bit and") makes it possible to obtain the "bit and" of two expressions.

4j3 Examples:

4j3a x

4j3b x .a y

4k Primary

4k1 Syntax: primary = bltin / abxreg / varfun / const /"(" exp ")" / immed / indir ;

4k1a bltin = ((".lrsh" "(" actual ")" .num /".lsh" "(" actual ")" .num / ".rsh" "(" actual ")" .num / ".rcy" "(" actual ")" .num / ".rcy" "(" actual ")" .num ) (",2" /.empty)) /".brs" .num "(" actual ")" ;

4k1b abxreg = ".ar" /".br" /".xr" ;

4k1c varfun = .id ("[" index "]" /"(" actual ")" /.empty);

4k1d const = .num;

4k1e immed = "$" (var / const ("[" index "]" /.empty));

4k1f indir = "[" (immed /var /const) "]" ;

4k1g var = .id ("[" index "]" /.empty);

4k1h index = "(" exp ")" / .num /(.id /".xr")("+" .num/"-" .num/.empty);

4k1i actual = ( .id / .empty ) $2( "," ( .id / .empty))

4k2 Semantics: The primary consists of the basic entities that can be used to construct an expression. It provides for direct reference to the A, B, and X registers, use of the shift and cycle instructions with optional tagging, use of the BRS instruction, indexed variables, functions of up to three arguments, and both indirect and immediate addressing. Note that by means of the parenthesis, recursion is introduced, and thus complex expressions may be constructed from simpler ones.

4k3 Examples:

4k3a x

4k3b x[i+1]

4k3c 23

4k3d pac(x,y)

4k3e (x + y)

4k3f [x]

4k3g $x

4k3h .lsh(m,0,6)3 + .rsh(a,b,x)5,2

5 Statements: A statement is the basic executable unit of an MOL program. It denotes some action that is to be performed, which action may be the evaluation of expressions or the execution of other statements.

5a Syntax: labeld = (parid ":" /.empty) stat ;

5a1 stat = if / simple ;

5a2 simple = block / goto / return / call / rcall / bump / arpas / iterat / entry / case / null / exu / assign ;.

5b If

5b1 Syntax: if = "if " bexp ("then " simple ("else " stat /.empty) /"do-single " stat);.

5b2 Semantics: The "if" construct is the standard if statement with the optional "else" part. The added construct "do-single" indicates that the true part will consist of just one instruction and thus the code at the end of the test for the "bexp" can be compiled to minimize the branch and skip instructions.

5b3 Examples:

5b3a if x then goto 12 else x←1;

5b3b if x .ne z do-single bump i;.

5c Block

5c1 Syntax: block = "begin " labeld $(";" labeld) "end";.

5c2 Semantics: The "block" construct allows the user to delimit a sequence of consecutive statements by "begin" and "end" to indicate that it is to be treated as a single statement. Note that declarations are not permitted within a block.

5c3 Examples:

5c3a begin x←1; y←x*y+z; (here): return(y) end;

5c3b begin call inchar(char); char←char .a 77b end;.

5d Goto

5d1 Syntax: goto = ("goto " /"go " "to ") addr ;

5d1a addr = var / indir / immed / const ;.

5d2 Semantics: The "goto" generates an unconditional branch. This branch can be indirect, indexed, direct, or immediate.

5d3 Examples:

5d3a go to here;

5d3b goto [$tra[i+1]];

5d3c goto $15b;.

5e Return

5e1 Syntax: "return" ('( actual ') /.empty ) ;.

5e2 Semantics: It is possible, via the "actual" construct, to indicate what the contents of the A, B and X registers should be when returning from a procedure. This is optional, and if nothing is specified the registers remain as affected by the procedure.

5e3 Examples:

5e3a return;

5e3b return(result);

5e3c return(m[i-2]-y,,m+1);.

5f Call

5f1 Syntax: "call " var ( '( actual ') / .empty ) ;.

5f2 Semantics: The optional arguments following the "call " indicate the contents of the A, B and X registers of the 940. Thus it is possible to pass up to 3 arguments at call time to a procedure. Also, it is possible to subscript the name of the procedure being called, thus indicating an alternate to the declared entry point.

5f3 Examples:

5f3a call sub;

5f3b call output(char .a 77b,,filen);

5f3c call table[i](arg1,10*arg2);.

5h Bump

5h1 Syntax: bump = "bump " addr $("," addr );.

5h2 Semantics: There is an instruction on the SDS 940 which adds 1 to memory, and leaves the contents of the central registers unchanged. The "bump " construct indicates that this operation is to be performed on the sequence of items that follow the "bump."

5h3 Examples:

5h3a bump i;

5h3b bump m[i-3],$1,[$stackp];.

5i Arpas

5i1 Syntax: arpas = "<" <copy across everything up to the next> ">" ;.

5i2 Semantics: This construct allows the user to insert machine code into an MOL program, if some special sequence of code that is needed cannot be generated or even expressed by the language.

5i3 Examples:

5i3a < sta temp>;

5i3b < cio fnumo; tco cr; tco lf; brs 10>;.

5j Iterat

5j1 Syntax: iterat = for / while;.

5k For

5k1 Syntax: for = "for " .id "from " exp ("inc " /"dec ") exp "to " exp "do " stat ;.

5k2 Semantics: The "for" statement provides a means of repeating a statement (or a block of statements) a specified number of times. By requiring the user to specify "inc" and "dec" it is possible to generate the appropriate code without complicated runtime or compile time computations. The limits on the for loop are not recomputed each time through the loop, but are computed once at the start. Note, however, that if an identifier is used as a limit, then the value of this identifier is used as the check each time, so that changing the value of this identifier will affect the "for" loop.

5k3 Examples:

5k3a for i from 1 inc 1 until n do 1[i]←0;

5k3b for j from x+1 inc 1 to x*x do begin n[j]←m[j+1]; m[j]←0 end;.

51 While

511 Syntax: while = "while " exp "do " stat ;.

512 Semantics: The "while" statement provides a means of repeating a statement (which can be a block) as long as an expression is true. This expression is reevaluated after each repetition of the "while" statement.

513 Examples:

513a while char .ne cr do char←inchar();

513b i←1; while i .le n do begin m[i]←0; bump i end;.

5m Entry

5m1 Syntax: entry = "entry " .id formal ;.

5m2 Semantics: The "entry" statement provides a means of indicating secondary entry points in a procedure. Any calling arguments that are indicated are stored, and a branch around the code generated by the "entry" statement is provided by the compiler, so that an "entry" statement can be inserted at any point without causing an interruption in the existing code.

5m2a The return address is moved from the entry point to the name of the procedure, so that all returns can return to the procedure name. However, this is not done in the case of a reentrant procedure, as the return address is placed elsewhere.

5m3 Examples:

5m3a entry subset;

5m3b entry inset(arg1,inch1);.

5n Case

5n1 Syntax: case = "case " exp "of " "begin " stat $('; stat ) "end";.

5n2 Semantics: The "case" statement provides a means of executing one statement out of many, depending on the value of the expression controlling the case statement. The same thing has usually been done by a series of nested "if" statements. If the value of the expression specifies a statement that does not lie within the range of the case statement, (i.e., from 1 to n=number of statements in the "case") then the last statement of the case is executed.

5n3 Examples:

```
5n3a case n of begin
call sub1(n);
return;
call error end;.
```

5o Null

5o1 Syntax: null = "null" ;.

5o2 Semantics: The "null" statement is included in the language so that there may be statements within the case statement which do nothing.

5p Execute

5p1 Syntax: exu = "execute " addr ;.

5p2 Semantics: This construct reflects the SDS 940 instruction which can execute another instruction. It provides a means of locating and executing this instruction with any appropriate address (i.e., with indirect addressing, index modification, etc.).

5p3 Examples:

5p3a execute m[i];

5p3b execute [$0];

5p3c execute 00220002b;.

5q Assign

5q1 Syntax: assign = (var /abxreg /indir / immed) $("," (var / abxreg / indir / immed)) '← ("+" /.empty ) exp ;.

5q2 Semantics: The "assign" statement provides a means of assigning values to variables, registers, and actual memory locations. Provision is made for multiple stores, in which case the stores are done in sequence from right to left. Also, if the item next to the ← is a register, the value will be placed in that register, and the remaining assignments done from that register; otherwise the assignments are taken from the register that the value happens to be left in by the expression analysis. Note too that the construct ←+ is used to indicate that an "add to memory" is to be done rather than a store. This is a special meaning, and thus precludes the use of a unary plus.

5q3 Examples:

5q3a x←1;

5q3b m[i],l←(x*b-c/d)+t;

5q3c .ar,m,.br←i+1;

5q3d m[i]←+.ar;.

1 The following is the syntax for the MOL. Note that backup is required to compile, but the backup is only past an identifier after the next character has been recognized. This gets over a lot of problems concerning assignment statements and labels.

```
2  prog = (.id  /.empty) $(arpas '; / proc ) "finish" ;

3  proc = parid ("pop" .sp  '(  .num   "," .num "," .num ') / .empty .rp
.rr) ("procedure" /"proc") formal '; (.tp  $decl2  /$declar) labeld $('; 
labeld) "endp." ;

   3a  parid = '( .id ') ;

   3b  formal = '( (.id ("," form1 / form4) /","   form1  /form4  ) ') /
   form4 ;

      3b1  form1 = .id ("," form2 / form3) /","  form2 /form3 ;

      3b2  form2 = .id /form3;

      3b3  form3 = .empty  ;

      3b4  form4 = .empty   ;

4  declar = (decl / decl2 ) '; ;

   4a  decl2 = ext / equ / virtue / frozen / prefix ;

   4b  decl = "declare" ("external" .rl /.empty .sl) item $("," item);

      4b1  item = .id (bound /.empty ) (value /.empty)  ;

      4b2  bound = "[" (.id /.num) "]" ;

      4b3  value = "=" ( '( icon $("," icon) ') / icon ) ;

      4b4  icon = (.num /.id /.st8  ) ;

   4c  ext = "external " evar $("," evar) ;

      4c1  evar = .id  ;

   4d  equ = "set " equ1 $("," equ1) ;

      4d1  equ1 = .id "=" (.id /.num ) ;

   4e  virtue = "virtual " cvar $("," cvar) ;

      4e1  cvar = .id  (bound /.empty ) ;
```

```
   4f  frozen = "frozen " frz1 $("," frz1) ;

      4f1  frz1 = .id ;

   4g   prefix   = "prefix "  "for "  ("generated "  "labels:"  .st8
   /"temporaries:" .st8 ) ;

5  labeld = (parid ":"  /.empty) stat ;

6  stat = if / simple ;

7  if = "if " bexp ("then " simple ("else " stat  / .empty ) /"do-single
" stat);

8  simple = block /  goto  /  return  /  call  / rcall / bump /  arpas /
iterat / entry / case / null / exu / assign ;

9  block = "begin " labeld $('; labeld) "end";

10  goto = ("goto " /"go " "to ") addr  ;

11  return = "return" ('( actual ')  /.empty ) ;

12  call = "call " var ( '( actual ') / .empty ) ;

13  bump = "bump " addr  $("," addr );

14  arpas = "<"  <copy across everything up to the next> ">" ;

15  iterat = for / while;

16  for = "for " .id "from " exp ("inc " .si  /"dec " .ri) exp "to " exp
"do " stat ;

17  while = "while " exp "do " stat ;

18  entry = "entry " .id formal ;

19  case = "case " exp "of " "begin " stat $('; stat ) "end" ;

20  null = "null" ;

21  exu = "execute " addr  ;

22  assign = (var /abxreg /indir / immed) $("," (var /  abxreg / indir /
immed)) '↑ ("+" .sa /.empty .ra) exp ;

23  exp = "if " bexp "then " bexp "else " exp / bexp;
```

```
24  bexp = union;

25  union = inter $("or " union );

26  inter = neg $("and " inter );

27  neg = "not " relat / relat;

28   relat  = sum (".lt " sum   .re .rb /".le " sum  .re .sb /".eq " sum
.re .rb /".ne  "  sum  .re .sb /".ge " sum  .re .sb /".gt " sum  .re .rb
/".cb " sum  .re .sb /".ncb " sum .re .rb /.empty);

29  sum = prod $("+" prod  /"-" prod ) ;

30  prod = factor $("*" factor /"/" factor /'← factor );

31  factor = bor /"-" factor  ;

32  bor = band $( ".v " band  / ".x " band );

33  band = prim $(".a " prim );

34  prim = bltin / abxreg / varfun .se / const .se / '( exp ') / immed /
indir;

35  abxreg = ".ar" / ".br" /".xr"  ;

36  bltin =((".lrsh" '( actual ') .num /".lsh" '( actual ') .num /".rsh"
'( actual ') .num )(",2" /.empty)) /".brs" .num '( actual ');

37  varfun = .id ("[" index "]" /'( actual ') /.empty );

38  var = .id ("[" index "]" / .empty);

39  index = '( exp ')  .te  /.num  /(.id  /".xr"  )  ("+" .num /"-" .num
/.empty ) ;

40  addr = var / indir / immed / const ;

41  immed = "$" (var / const ("[" index "]" / .empty)) ;

42  indir = "[" (immed /var /const ) "]"  ;

43  const = .num .se ;

44  actual = .empty (exp ("," act1 / act4) /","  act1 /.empty  act4 ) ;

   44a  act1 = exp ("," act2 / act3) /","  act2 /act3 ;
```

```
   44b  act2 = exp /act3 ;
   44c  act3 = .empty  ;
   44d  act4 = .empty   ;
45  synerr = $("endp." /);  .end
```

# 1 The MOL Executive

## 1a User Interface

1a1 The MOL Executive is the interface between the user and the MOL compiler. It uses the command-recognition structure of the SDS 940 time sharing system itself, especially that of the QED subsystem.

1a1a A special meaning is attached to certain control characters; when one of them is typed by the user, the remainder of the control word or phrase is echoed by the EXEC. Some characters represent commands to be performed, others represent flags requiring a yes/no type of answer, and others require file names, such as Input:/prog/.

1a1b Each command requires a period for confirmation. If any other character is typed, then a space and a question mark are echoed and the command is aborted.

1a2 The various characters recognized and their meanings are as follows:

1a2a (i) Input: "I" is typed to specify the input file for the MOL compiler. After the I has been typed, a file name should be given, followed by a period.

1a2a1 An input file must be specified with each new compilation. This file will be closed when the compilation is finished.

1a2b (o) Output: "OS is typed to specify the output file for the MOL compiler. After the "O" has been typed, a file name is expected and should be acknowledged by a period.

1a2b1 Each time the compilation process is initiated, the old output file is closed and the new one opened. If, however, the new output file name is the same as the last one used for output, or if none has been specified, then the last file is not closed and the next set of output is appended to the current output file.

1a2b2 It is possible to specify different files for output, should the wrong one be given. However, when execution of the compiler begins, the last file specified for output will be used.

1a2c (b) Begin Compilation: "B" is typed to indicate that all file names and flags have been specified for the current

compilation, so that compilation may now actually be initiated.

1a2c1 If there is insufficient information (such as lack of file names) to initiate the compilation process, the command will be aborted.

1a2c2 When a successful compilation has been performed, the message "***end of compilation***" is typed. If control returns to the user without this message, then the compilation has not been completed because of an error condition (such as running out of room on the RAD, or an illegal instruction trap from the compiler, etc.).

1a2d (z) Zap: "Z" is typed to terminate the MOL Executive and return control to the TSS Executive. When "zap." is typed, any remaining files that are open are closed.

1a2e (l) Listing (interlinear): "L" is typed to set the flag controlling the interlinear listing. The expected response is either a "y" or "n" for "yes" and "no", respectively, although a period alone will be taken as a "yes" response.

1a2e1 When the interlinear listing is sent to any file other than the controlling Teletype, all semicolons are converted into $ so that ARPAS will not terminate a comment in the middle of the line.

1a2f (t) Type Procedure Names: "T" is typed to set the flag which determines whether or not procedure names are typed on the controlling Teletype as they are compiled. If the flag is set, then as each procedure is encountered by the compiler, the name of the procedure is typed. The response to this command is in the usual "y" (yes) or "n" (no) manner.

1a2g (c) Cross Reference: "C" is typed to request a cross-reference listing of the identifiers used in the input file. The response to this command is a file name that is to be used for the cross-reference listing, such as "Teletype".

1a2g1 This listing gives the names of the identifiers in alphabetical order, along with their status (undefined, not used, etc.) and an ordered list of the line numbers on which they are used.

1a2h (r) Reentrant: "R" is typed to set the flag that governs whether or not the compilation produces reentrant code. A "y" or "n" response for "yes" or "no" is expected and must be acknowledged with a period.

1a2h1 If the response if yes, then the flag for "generate temporaries" (see below) is automatically set to "no".

1a2i (g) Generate Temporaries: "G" is used to set the flag which specifies whether or not the temporaries used in the last input file are to be allocated at the end of the output file.

1a2i1 If this flag is on, the the temporaries are allocated (this is the usual case). If the flag is off (set by giving a "no" response), then the temporaries are not allocated. The latter is generally used when reentrant code is being produced, and then in connection with the "prefix for temporaries " declaration.

1a2j (k) Keep compiling: "K" is the same as "begin compiling," except that some parts of the MOL compiler are not reinitialized:

1a2j1 These are the symbol table and the temporary- and generated-label counts. The purpose of this command is to provide a means of compiling one input file, and then another, as if they were all the same input file.

1a2k (q) Quick: "Q" causes the supression of the string which is normally echoed for each command character.

1a2l (v) Verbose: "V" causes the printing of the string which gives the meaning for each character typed as a command.

1a2m Any other characters typed are illegal; the MOL Executive will respond with a space followed by a question mark.

1b Error Recovery and Error Messages

1b1 The only errors which should normally be expected are syntax errors in the user's input file.

1b1a When such an error occurs, an appropriate error message is typed, along with the line number and line which caused the error. Also an uparrow is typed under the last character interpreted by the compiler.

1b1b To attempt an error recovery, a scan is made for the next "endp.", stacks are reset, and an attempt is made to restart the compiler to look for a procedure. This type of procedure has proven fairly useful, and is far better than just giving up.

1b2 Another user error which may arise is the occurrence of

identifiers or numbers longer than the maximum length allowed (6 and 9 respectively). In this case a warning message is typed, the remainder of the string is skipped, and compilation continues.

1b3 Next on the list of errors are stack and symbol-table over/underflow.

1b3a All the stacks and symbol tables have been set to adequate sizes for most programs, and the normal user will never encounter the bounds. When and if they are exceeded, an error message to this effect is typed and the compilation process is terminated.

1b4 Yet another, even more obscure, error is one caused by an illegal string passed to FMT (a routine internal to the MOL compiler).

1b4a Such a string originates in the syntax equations themselves, and this error can only be the result of changes made in the syntax file of the compiler; when this is cross-checked by FMT, the error is detected. This is treated as a fatal error, and compilation ceases. But this error should never occur in the normal course of events.

1b5 Finally there are two types of errors from which there is no recovery at present.

1b5a Internal conditions in the compiler, such as illegal memory references or illegal instructions, or program loops (hopefully none of these will ever occur).

1b5b Conditions external to the compiler, such as running out of room on the RAD, or a rubout by the user, or a system crash.

1 (inchar) The "inchar" procedure is an intermediate interface between the input medium and the compiler.

1a This routine buffers one line of text at a time, outputs it to the output file (if the list option is set) and returns the next character in the A register.

1b "inchar" also has an entry point to print error comments to the controlling Teletype should any syntax error be detected.

```
(inchar) procedure;
  declare nchar=80, mchar=80, maxch=80, line[80], i ;
  declare external list=1, nline=0, lf=153b, cr=155b, space=0b;
  declare star='  *',arrow='  ↑', peeked=0;
  if peeked then
    begin
      peeked←0;
      return(line[nchar]) end;
  if nchar .ge mchar
    then
      begin
        for i from 0 inc 1 to maxch do
          begin
            line[i] ← gench();
            if .ar .eq lf then goto m1    end;
        mchar ← maxch;
        goto m2;
        (m1): mchar ← i;
        (m2): if list then
          begin
            call putch(star);
            for i from 0 inc 1 to mchar do call putch(line[i]) end;
        nchar ← 0;
        bump nline end
    else bump nchar;
  return(line[nchar]);
  entry (perr);
    call putch(star);
    for i from 0 inc 1 to mschar do putch(line[i]);
    for i from 0 inc 1 to nchar-1 do putch(space);
    call putch(arrow);
    call putch(cr);
    call putch(lf);
    return
  endp.
```

1 (Book1) E. Book and D. V. Schorre, "A Higher-Level Machine-Oriented Language as an Alternative to Assembly Langauge," Tech Memo 3086/001/00, System Development Corporation.

2 (Book2) E. Book and D.V. Schorre, "A User's Manual for MOL-360", Tech Memo 3086/003/00, System Development Corporation.

3 (Wirth1) N. Wirth, "PL360, a Programming Language for the 360 Computers," Journal ACM (January 1968).

4 (Wirth2) N. Wirth and H. Weber, "EULER: A Generalization of ALGOL, and its Formal Definition," Comm. ACM (January-February 1966).

5 (Wirth3) N. Wirth and C. A. R. Hoare, "A Contribution to the Development of ALGOL," Comm. ACM (June 1966).

6 (Schorre1) D. V. Schorre, "Improved Organization for Procedural Languages," Tech Memo 3086/002/00, System Development Corporation.

7 (Engelbart1) D. C. Engelbart, "Study for the Development of Human Intellect Augmentation Techniques," Final Report, Contract NAS 1-5904, SRI Project 5890, Stanford Research Institute, Menlo Park, California (in preparation).

OTHER AHI REPORTS


1 D. C. Engelbart, "Special Considerations of the Individual as A User, Generator, and Retriever of Information," Paper presented at the Annual Meeting of American Documentation Institute, Berkeley, California, (23-27 October 1960)

2 D. C. Engelbart, "Augmenting Human Intellect: A Conceptual Framework," Summary Report, Contract AF 49(638)-1024, SRI Project 3578, Stanford Research Institute, Menlo Park, California (October 1962) (AD289565)*

3 D. C. Engelbart, "A Conceptual Framework for the Augmentation of Man's Intellect," in Vistas in Information Handling, Volume 1, D. W. Howerton and D. C. Weeks, eds., Spartan Books, Washington, D.C. (1963)

4 D. C. Engelbart, "Augmenting Human Intellect: Experiments, Concepts, and Possibilities," Summary Report, Contract AF 49(638)-1024, SRI Project 3578, Stanford Research Institute, Menlo Park, California (March 1965) (AD640989)*

5 D. C. Engelbart and B. Huddart, "Research on Computer-Augmented Information Management," Technical Report No. ESD-TDR-65-168, Contract AF 19(628)-4088, Stanford Research Institute, Menlo Park, California (March 1965) (AD622520)*

6 W. K. English, D. C. Engelbart and B. Huddart, "Computer-Aided Display Control," Final Report, Contract NAS 1-3988, SRI Project 5061, Stanford Research Institute, Menlo Park, California (July 1965)**

7 D. C. Engelbart, "Study for the Development of Human Intellect Augmentation Techniques," Interim Progress Report, Contract NAS 1-5904, SRI Project 5890, Stanford Research Institute, Menlo Park, California (March 1967)


* Reports with AD numbers are available from Defense Documentation Center, Building 5, Cameron Station, Alexandria Virginia 22314.

** Reference No. 6 may be obtained from CFSTI, Sills Building, 5825 Port Royal Road, Springfield, Virginia 22151; cost $3.00 per copy or 75 cents for microfilm.

UNCLASSIFIED

Security Classification

## DOCUMENT CONTROL DATA - R & D

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

| 1. ORIGINATING ACTIVITY *(Corporate author)* | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| Stanford Research Institute<br>333 Ravenswood Avenue<br>Menlo Park, California 94025 | Unclassified |
| | 2b. GROUP<br>N/A |

3. REPORT TITLE

MOL940: PRELIMINARY SPECIFICATION FOR AN ALGOL-LIKE MACHINE-ORIENTED LANGUAGE FOR THE SDS 940

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Interim Technical Report 2

5. AUTHOR(S) *(First name, middle initial, last name)*

R. E. Hay

J. F. Rulifson

| 6. REPORT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| March 1968 | 35 | 14 |

| 8a. CONTRACT OR GRANT NO. | 9a. ORIGINATOR'S REPORT NUMBER(S) |
|---|---|
| NAS1-5904<br>b. PROJECT NO. | SRI Project 5890<br>Interim Technical Report 2 |
| c.<br>d. | 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* |

10. DISTRIBUTION STATEMENT

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY |
|---|---|
| | NASA-Langley Research Center<br>Mail Stop 126, Langley Station<br>Langley, Virginia 23365 |

13. ABSTRACT

This report is a reference manual for a programming language developed at Stanford Research Institute for the Scientific Data Systems 940 computer. The compiler is now fully operational; it is written in its own language, compiles itself, and is in daily use for development of our CRT-display service system.

The name MOL940 (or simply MOL) is an acronym for "Machine-Oriented Language." MOL is an ALGOL-like language with natural extensions for bit manipulation. The added syntax strongly reflects the internal design of the SDS 940, in accordance with the name MOL.

The introduction to this report includes a brief summary of other projects of the same nature which were known to the authors. There is also a discussion of the design criteria that shaped the MOL. The major topics are the comprehensibility of programs written in the language, the needs of system programmers working within a time-sharing system, and the effects on coding that result from using an on-line CRT.

A complete definition of the language is given, using an extended Backus Normal form; included are semantic explanations and examples, a sample program, and some examples of code produced by the MOL compiler.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

UNCLASSIFIED

Security Classification

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| Programming | | | | | | |
| Compiler | | | | | | |
| High-Level Language | | | | | | |
| Machine-Oriented Language | | | | | | |