/MDLR, 08/30/68 1557:55 JFR ; .MCH=72; .MLN=52; .PLO=1;

Co-routines

```
| .HED="MOL - ABSTRACT"; |.PGN=0; .ROM=I; .RES;
```

MOL - ABSTRACT

1A 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.

1B 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.

1C 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.

1D 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.

2 .hed="MOL - CONTENTS"; .PGN=.PGN-1; .RES;

MOL - ABSTRACT

3 .HED="MOL - FOREWORD"; .PGN=.PGN-1; .RES;

update

3A Development of the MOL (Machine-Oriented Language) initially began in October 1966 under ARNAS-NASA sponsorship. Although completion took approximately one year, only six man months have been invested in the project. The Augmented Human Intellect (AHI) Program (ENGELBART7) 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.

3B This report has been prepared with the On-Line Text Manipulation System, and consequently it differs 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.

this is new version

 .HED="MOL - INTRODUCTION"; .PGN=0; .ROM=0; .RES;

4A 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.

4A1 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.

4B 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.

4C 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.

4C1 Algorithmic clarity is mainly due to the structure implicit in the syntax of the language.

4C1A 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).

4C1B 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.

4C2 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.

4C2A 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.

4C3 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.

4C4 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.

4C4A 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.

4C4B 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.

4C4C 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 ESDI we presented some basic discussion in this direction (See ENGELBART5).

?

4D 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.

5 .HED="MOL - DEFINITIONS"; .PGN=.PGN-1; .RES;

5A Terminology

5A1 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.

5A1A 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.

5A1B 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).

5A1B1 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.

5A1C Concatenation is designated by writing items consecutively. The items are separated by slashes to indicate alternation. Each syntax equation ends with a semicolon.

5A1D 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.

5A1E 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.

5A1F 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:

5A1F1 identifier = letter $(digit / letter);.

5A2 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:

5A2A Most of the people using and designing the MOL language and compiler have had direct experience writing recursive recognizer compilers.

5A2B 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.

5B Basic Symbols and Syntactic Entities

5B1 General Vocabulary

5B1A Terminal Vocabulary

5B1A1 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 ( ) + ' $ * : = - + ; ↑ , . / < > [ ]

5B1A2 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

5B1B Nonterminal Vocabulary

5B1B1 <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>

5B2 Primitives

5B2A 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.

5B2A1 Syntax: id = letter $5(letter / digit);.

5B2A2 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.

5B2A2A 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.

5B2A3 Examples of Identifiers:

5B2A3A I

5B2A3B CHAR

5B2A3C X21

5B2A3D I2JBY1

5B2B Numbers

5B2B1 Syntax: number = 1$8 digit ("b" / .empty) ;.

5B2B2 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.

5B2B3 Examples of Numbers:

5B2B3A 1

5B2B3B 1024

5B2B3C 77770000b

5B2C Strings

5B2C1 8-bit character strings are the only strings recognized by the MOL compiler, and these can only occur in declarations.

5C 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.

5C1 Procedure: The procedure is the basic syntactic entity, in that one writes procedures, which are compiled, assembled, and loaded.

5C1A Syntax: procedure = parid ("pop" "(" .num "," .num "," .num ")" /.empty) ("procedure" /"proc") formal ";" $declar labeld $(";" labeld) "endp.";

5C1A1 parid = "(" id ")" ;

5C1A2 formal = "(" (id / .empty) $2("," (id / .empty)) ")" ;.

5C1B 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.

5C1B1 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.

5C1B2 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.

5C1B3 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.

5C1B4 Finally, all procedures must end with an "endp".

5C1C Example of Procedure:

5C1C1 (get) procedure(x,i); declare x,i; return([x[i+1]]+4) endp.

5C2 Declaration

5C2A Syntax: declaration = (decl / ext / equ / virtue / frozen / prefix) ";" ;.

5C3 Decl

5C3A Syntax: decl = "declare " ("external " /.empty) item $("," item);

5C3A1 item = .id (bound /.empty) (value /.empty) ;

5C3A2 bound = "[" (.id /.num) "]" ;

5C3A3 value = "="("(" icon $("," icon) ")" / icon);

5C3A4 icon = (.num /.id /.st8) ;.

5C3B 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).

5C3C Examples:

5C3C1 declare x,y,z[10];

5C3C2 declare external m=10,n=m,st='end of file';

5C3C3 declare sk[10]=(0,1,20,40);.

5C4 External

5C4A Syntax: ext = "external " evar $("," evar);

5C4A1 evar = .id ;.

5C4B 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.

5C4C Example

5C4C1 external m,n,z;.

5C5 Equate

5C5A Syntax: equ = "set " equl $("," equl) ;

5C5A1 equl = .id "=" (.id /.num) ;.

5C5B 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.

5C5C Example:

5C5C1 set m940=1,skmax=100;.

5C6 Virtual

5C6A Syntax: virtue = "virtual " cvar $("," cvar) ;

5C6A1 cvar = .id (bound /.empty);.

5C6B 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.

5C6C Example:

5C6C1 virtual a,b,m[32b];.

5C7 Frozen

5C7A Syntax: frozen = "frozen " frzl $("," frzl);

5C7A1 frzl = .id;.

5C7B 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.

5C7C Example:

5C7C1 frozen a,b,x;.

5C8 Prefix

5C8A Syntax: prefix = "prefix " "for " ("generated " "labels:" .st8 /"temporaries:" .st8) ;.

5C8B 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.

5C8C Examples:

5C8C1 prefix for generated labels: 'fmt';

5C8C2 prefix for temporaries: 'libet' ;.

5D 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.

5D1 Exp

5D1A Syntax: exp = "if" bexp "then" bexp "else" exp / bexp ;.

5D1B 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.

5D1C Examples:

5D1C1 if x .le y then 1 else 2

5D1C2 x+y*z/(x+1)

5D2 Bexp

5D2A Syntax: bexp = union;.

5D3 Union

5D3A Syntax: union = intersection $("or" union);.

5D3B 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.

5D3C Example:

5D3C1 x or y

5D4 Intersection

5D4A Syntax: intersection = negation $("and" intersection);.

5D4B 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.

5D4C Example:

5D4C1 x and y

5D5 Negation

5D5A Syntax: negation = "not" negation / relation;.

5D5B Semantics: This construct makes it possible to take the (logical) negation of the value of any expression.

5D5C Example:

5D5C1 not x

5D6 Relation

5D6A Syntax: relation = sum (".gt" sum /".ge" sum /".ne" sum /".eq" sum /".le" sum /".lt" sum /".cb" sum /".ncb" sum /.empty);

5D6B 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.

5D6C Examples:

5D6C1 m .gt n

5D6C2 z .ne y

5D6C3 x .cb y

5D7 Sum

5D7A Syntax: sum = product $("+" product / "-" product);.

5D7B Semantics: The sum permits one to combine expressions with the arithmetic operators + and - . Note that all values are taken to be 24-bit integers.

5D7C Examples:

5D7C1 x

5D7C2 x + y

5D7C3 x - y + z

5D8 Product

5D8A Syntax: product = factor $("*" factor / "/" factor / "↑" factor);.

5D8A1 Syntax: factor = bor / "-" factor ;

5D8B 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.

5D8C Examples:

5D8C1 x

5D8C2 x * y

5D8C3 x / y

5D8C4 x ↑ y

5D9 Bor

5D9A Syntax: bor = band $(".v" band / ".x" band);.

5D9B 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.

5D9C Examples:

5D9C1 x

5D9C2 x .v y

5D9C3 x .x y

5D10 Band

5D10A Syntax: band = primary $(".a" primary);.

5D10B Semantics: The "band" (standing for "bit and") makes it possible to obtain the "bit and" of two expressions.

5D10C Examples:

5D10C1 x

5D10C2 x .a y

5D11 Primary

5D11A Syntax: primary = bltin / abxreg / varfun / const /"(" exp ")" / immed / indir ;

5D11A1 bltin = ((".lrsh" "(" actual ")" .num /".lsh" "(" actual ")" .num / ".rsh" "(" actual ")" .num / ".rcy" "(" actual ")" .num / ".rcy" "(" actual ")" .num) (",2" /.empty)) / ".brs" .num "(" actual ")" ;

5D11A2 abxreg = ".ar" /".br" /".xr" ;

5D11A3 varfun = .id ("[" index "]" /"(" actual ")" /.empty);

5D11A4 const = .num;

5D11A5 immed = "$" (var / const ("[" index "]" /.empty));

5D11A6 indir = "[" (immed /var /const) "]" ;

5D11A7 var = .id ("[" index "]" /.empty);

5D11A8 index = "(" exp ")" / .num /(.id /".xr")("+" .num/"-" .num/.empty);

5D11A9 actual = (.id / .empty) $2("," (.id / .empty))

5D11B 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.

5D1IC Examples:

5D1IC1 x

5D1IC2 x[i+1]

5D1IC3 23

5D1IC4 pac(x,y)

5D1IC5 (x + y)

5D1IC6 [x]

5D1IC7 $x

5D1IC8 .lsh(m,0,6)3 + .rsh(a,b,x)5,2

5E 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.

5E1 Syntax: labeld = (parid ":" /.empty) stat ;

5E1A stat = if / simple ;

5E1B simple = block / goto / return / call / rcall / bump / arpas / iterat / entry / case / null / exu / assign ;.

5E2 If

5E2A Syntax: if = "if " bexp ("then " simple ("else " stat /.empty) /"do-single " stat);.

5E2B 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.

5E2C Examples:

5E2C1 if x then goto 12 else x←1;

5E2C2 if x .ne z do-single bump i;.

5E3 Block

5E3A Syntax: block = "begin " labeld $(";" labeld) "end";.

5E3B 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.

5E3C Examples:

5E3C1 begin x←1; y←x*y+z; (here): return(y) end;

5E3C2 begin call inchar(char); char←char .a 77b end;.

5E4 Goto

5E4A Syntax: goto = ("goto " /"go " "to ") addr ;

5E4A1 addr = var / indir / immed / const ;.

5E4B Semantics: The "goto" generates an unconditional branch. This branch can be indirect, indexed, direct, or immediate.

5E4C Examples:

5E4C1 go to here;

5E4C2 goto [$tra[i+1]];

5E4C3 goto $15b;.

5E5 Return

5E5A Syntax: "return" ('(actual ') /.empty) ;.

5E5B 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.

5E5C Examples:

5E5C1 return;

5E5C2 return(result);

5E5C3 return(m[i-2]-y,,m+1);.

5E6 Call

5E6A Syntax: "call " var ('(actual ') / .empty) ;.

5E6B 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.

5E6C Examples:

5E6C1 call sub;

5E6C2 call output(char .a 77b,,filen);

5E6C3 call table[i](arg1,10*arg2);.

5E7 Bump

5E7A Syntax: bump = "bump " addr $("," addr);.

5E7B 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."

5E7C Examples:

5E7C1 bump i;

5E7C2 bump m[i-3],$1,[$stackp];.

5E8 Arpas

5E8A Syntax: arpas = "<" <copy across everything up to the next> ">" ;.

5E8B 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.

5E8C Examples:

5E8C1 < sta temp>;

5E8C2 < cio fnumo; tco cr; tco lf; brs 10>;.

5E9 Iterat

5E9A Syntax: iterat = for / while;.

5E10 For

5E10A Syntax: for = "for " .id "from " exp ("inc " / "dec ") exp "to " exp "do " stat ;.

5E10B 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.

5E10C Examples:

5E10C1 for i from 1 inc 1 until n do l[i]←0;

5E10C2 for j from x+1 inc 1 to x*x do begin m[j]←m[j+1]; m[j]←0 end;.

5E11 While

5E11A Syntax: while = "while " exp "do " stat ;.

5E11B 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.

5E11C Examples:

5E11C1 while char .ne cr do char←inchar();

5E11C2 i←1; while i .le n do begin m[i]←0; bump i end;.

5E12 Entry

5E12A Syntax: entry = "entry " .id formal ;.

5E12B 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.

5E12B1 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.

5E12C Examples:

5E12C1 entry subset;

5E12C2 entry inset(arg1,inch1);.

5E13 Case

5E13A Syntax: case = "case " exp "of " "begin " stat $('; stat) "end";.

5E13B 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.

5E13C Examples:

5E13C1 case n of begin .crl; call subl(n); .crl; return; .crl; call error end;.

5E14 Null

5E14A Syntax: null = "null" ;.

5E14B Semantics: The "null" statement is included in the language so that there may be statements within the case statement which do nothing.

5E15 Execute

5E15A Syntax: exu = "execute " addr ;.

5E15B 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.).

5E15C Examples:

5E15C1 execute m[i];

5E15C2 execute [$0];

5E15C3 execute 00220002b;.

5E16 Assign

5E16A Syntax: assign = (var /abxreg /indir / immed) $("," (var / abxreg / indir / immed)) '← ("+" /.empty) exp ;.

5E16B 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.

5E16C Examples:

5E16C1 x←1;

5E16C2 m[i],1←(x*b-c/d)+t;

5E16C3 .ar,m,.br←i+1;

5E16C4 m[i]←+.ar;.

6 .HED="MOL - SYNTAX"; ⊢.PGN=.PGN-1; .RES;

MOL - SYNTAX

6A 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.

```
6B prog = (.id /.empty) $(arpas '; / proc ) "finish" ;

6C  proc = parid ("pop" .sp '( .num "," .num "," .num ') / .empty .rp
.rr) ("procedure"  /"proc")  formal  ';  (.tp $decl2 /$declar) labeld
$('; labeld) "endp." ;

    6C1 parid = '( .id ') ;

    6C2 formal = '( (.id ("," forml / form4)  /"," forml /form4 ) ') /
    form4 ; 6C2A forml = .id ("," form2 / form3) /"," form2 /form3 ;

        6C2A form2 = .id /form3;

        6C2B form3 = .empty ;

        6C2C form4 = .empty ;

6D declar = (decl / decl2 ) '; ;

    6D1 decl2 = ext / equ / virtue / frozen / prefix ;

    6D2  decl  =  "declare" ("external" .rl /.empty  .sl)  item  $(","
    item);

        6D2A item = .id (bound /.empty ) (value /.empty) ;

        6D2B bound = "[" (.id /.num) "]" ;

        6D2C value = "=" ( '( icon $("," icon) ') / icon ) ;

        6D2D icon = (.num /.id /.st8 ) ;

    6D3 ext = "external " evar $("," evar) ;

        6D3A evar = .id ;

    6D4 equ = "set " equl $("," equl) ;

        6D4A equl = .id "=" (.id /.num ) ;

    6D5 virtue = "virtual " cvar $("," cvar) ;

        6D5A cvar = .id (bound /.empty ) ;

    6D6 frozen = "frozen " frzl $("," frzl) ;
```

```
6D6A frzl = .id ;

6D7 prefix = "prefix " "for " ("generated " "labels:" .st8
/"temporaries:" .st8 ) ;

6E labeld = (parid ":" /.empty) stat ;

6F stat = if / simple ;

6G if = "if " bexp ("then " simple ("else " stat / .empty )
/"do-single " stat);

6H simple = block / goto / return / call / rcall / bump / arpas /
iterat / entry / case / null / exu / assign ;

6I block = "begin " labelc $('; labeld) "end";

6J goto = ("goto " /"go " "to ") addr ;

6K return = "return" ('( actual ') /.empty ) ;

6L call = "call " var ( '( actual ') / .empty ) ;

6M bump = "bump " addr $("," addr );

6N arpas = "<" <copy across everything up to the next> ">" ;

6O iterat = for / while;

6P for = "for " .id "from " exp ("inc " .si /"dec " .ri) exp "to "
exp "do " stat ;

6Q while = "while " exp "do " stat ;

6R entry = "entry " .id formal ;

6S case = "case " exp "of " "begin " stat $('; stat ) "end" ;

6T null = "null" ;

6U exu = "execute " addr ;

6V assign = (var /abxreg /indir / immed) $("," (var / abxreg / indir
/ immed)) '+ ("+" .sa /.empty .ra) exp ;

6W exp = "if " bexp "then " bexp "else " exp / bexp;

6X bexp = union;

6Y union = inter $("or " union );
```

```
6Z inter = neg $("and " inter );

6AA neg = "not " relat / relat;

6AA 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);

6AB sum = prod $("+" prod /"-" prod ) ;

6AC prod = factor $("*" factor /"/" factor /'+ factor );

6AD factor = bor /"-" factor ;

6AE bor = band $( ".v " band / ".x " band );

6AF band = prim $(".a " prim );

6AG prim = bltin / abxreg / varfun .se / const .se / '( exp ") /
immed / indir;

6AH abxreg = ".ar" / ".br" /".xr" ;

6AI bltin =(((".lrsh" '( actual ') .num /".lsh" '( actual ') .num
/".rsh" '( actual ') .num )(",2" /.empty)) /".brs" .num '( actual ');

6AJ varfun = .id ("[" index "]" /'( actual ') /.empty );

6AK var = .id ("[" index "]" / .empty);

6AL index = '( exp ') .te /.num /(.id /".xr" ) ("+" .num /"-" .num
/.empty ) ;

6AM addr = var / indir / immed / const ;

6AN immed = "$" (var / const ("[" index "]" / .empty)) ;

6AO indir = "[" (immed /var /const ) "]" ;

6AP const = .num .se ;

6AQ actual = .empty (exp ("," act1 / act4) /"," act1 /.empty act4 ) ;
6AR1 act1 = exp ("," act2 / act3) /"," act2 /act3 ;

    6AQ1 act2 = exp /act3 ;

    6AQ2 act3 = .empty ;

    6AQ3 act4 = .empty ;
```

```
6AR synerr = $("endp." /); .end
```

7 .HED="MOL - OPERATION"; .PGN=.PGN-1; .RES;

7A User Interface

7A1 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.

7A1A 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/.

7A1B 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.

7A2 The various characters recognized and their meanings are as follows:

7A2A (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.

7A2A1 An input file must be specified with each new compilation. This file will be closed when the compilation is finished.

7A2B (o) Output: "O" 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.

7A2B1 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.

7A2B2 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.

7A2C (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.

7A2C1 If there is insufficient information (such as lack of file names) to initiate the compilation process, the command will be aborted.

7A2C2 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.).

7A2D (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.

7A2E (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.

7A2E1 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.

7A2F (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.

7A2G (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".

7A2G1 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.

7A2H (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.

7A2H1 If the response if yes, then the flag for "generate temporaries" (see below) is automatically set to "no".

7A2I (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.

7A2I1 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.

7A2J (k) Keep compiling: "K" is the same as "begin compiling," except that some parts of the MOL compiler are not reinitialized:

7A2J1 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.

7A2K (q) Quick: "Q" causes the supression of the string which is normally echoed for each command character.

7A2L (v) Verbose: "V" causes the printing of the string which gives the meaning for each character typed as a command.

7A2M Any other characters typed are illegal; the MOL Executive will respond with a space followed by a question mark.

7B Error Recovery and Error Messages

7B1 The only errors which should normally be expected are syntax errors in the user's input file.

7B1A 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.

7B1B 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.

7B2 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.

7B3 Next on the list of errors are stack and symbol-table over/underflow.

7B3A 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.

7B4 Yet another, even more obscure, error is one caused by an illegal string passed to FMT (a routine internal to the MOL compiler).

7B4A 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.

7B5 Finally there are two types of errors from which there is no recovery at present.

7B5A Internal conditions in the compiler, such as illegal memory references or illegal instructions, or program loops (hopefully none of these will ever occur).

7B5B Conditions external to the compiler, such as running out of room on the RAD, or a rubout by the user, or a system crash.

```
8 .HED="MOL - SAMPLE PROGRAM"; .PGN=.PGN-1; .RES;
```

8A (inchar) The "inchar" procedure is an intermediate interface between the input medium and the compiler.

8A1 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.

8A2 "inchar" also has an entry point to print error comments to the controlling Teletype should any syntax error be detected. .dsn=1; .lsp=0; .min=28; .ins=2;

```
8B (inchar) procedure; .scr=1;

   8B1 declare nchar=80, mchar=80, maxch=80, line[80], i ;

   8B2 declare external list=1, nline=0, lf=153b, cr=155b, space=0b;

   8B3 declare star=' *',arrow=' ↑', peeked=0;

   8B4 if peeked then

      8B4A begin

         8B4A1 peeked←0;

         8B4A2 return(line[nchar]) end;

   8B5 if nchar .ge mchar

      8B5A then

         8B5A1 begin

            8B5A1A for i from 0 inc 1 to maxch do

            8B5A1A1 begin

            8B5A1A1A line[i] ← gench();

            8B5A1A1B if .ar .eq lf then goto m1 end;

            8B5A1B mchar ← maxch;

            8B5A1C goto m2;

            8B5A1D (m1): mchar ← i;

            8B5A1E (m2): if list then

            8B5A1E1 begin
```

```
8B5A1E1A call putch(star);

8B5A1E1B for i from 0 inc 1 to mchar do call
putch(line[i]) end;

8B5A1F nchar ← o;

8B5A1G bump nline end

8B5B else bump nchar;

8B6 return(line[nchar]);

8B7 entry (perr);

8B7A call putch(star);

8B7B for i from 0 inc 1 to mschar do putch(line[i]);

8B7C for i from 0 inc 1 to nchar-1 do putch(space);

8B7D call putch(arrow);

8B7E call putch(cr);

8B7F call putch(lf);

8B7G return

8B8 endp.
```

```
9 .HED="MOL - COMPILER LISTING"; .PGN=.PGN-1; .RES;
```

MOL - COMPILER LISTING

```
9A %mol % .meta prog (k=100,m=100,n=100,ss=200)

9B %parse rules%

   9B1 %file and proedure headings% need add reentrant coand generate
   temp options

      9B1A prog = (.id /.empty) $(arpas '; / proc) "finish" &;

      9B1B proc =

         9B1B1 parid ("pop" '( sinum ', sinum ', sinum ') /.empty
         ("procedure"/"proc") formal ';

         9B1B2 $( declar ';)

         9B1B3 labeled $('; labeled)

         9B1B4 "endp." &;

      9B1C parid = ('( /.empty) .id (') /.empty);

      9B1D labeled = (+parid (': / "::") /.empty) stat &;

   9B2 %declarctions%

      9B2A declar = decl / ext / equ / virtue / frozen / prefix;

      9B2B decl = "declare " ("external" :ext / .empty :mt)[0] item
      $(', item :do[2]) :dcdecl[2];

         9B2B1 item = .id (bound /.empty :mt[0]) (value /.empty
         :mt[0]) :itm[3];

         9B2B2 bound = '[ (.num / .id) '] :bnd[1];

         9B2B3 value = '= ( '( icon $(', icon :do[2]) ') / icon)
         :val[1];

         9B2B4 icon = sinum / .id / .sr ;

      9B2C ext = "external " .id (', .id :do[2]) :cext[1];

      9B2D equ = "set " equl (', equl :do[2]) :cequ[1];

         9B2D1 equl = .id '= (.id / sinum) :equs[2];

      9B2E virtue = "virtual " cvar $(', cvar :do[2]) :cvirtu[1];

         9B2E1 cvar = .id (bound / .empty :mt[0]) :ccvar[2];
```

array pipes

Pipes & Co-routines

```
9B2F frozen = "frozen " .id $(', .id )&; %this is going to go!%

9B2G prefix = "prefix " "for" %will go also, but need abity to
set tempts to a unknown symbol%

   9B2G1 ("generated" "labels" .sr !"set it now" /

   9B2G2 "temporaries" .sr !"and this too");

9B3 stat = (if / simple) * &;

   9B3A if = "if " bexp

      9B3A1 "then" *1:if1[2]* simple (

         9B3A1A "else" *1*2:if[2]* stat *2:bru[1]* /

         9B3A1B .empty) /

      9B3A2 "do-single" stat ) *1def[1];

   9B3B simple = block /branch /suber /iterat /case /other /exp &;

      9B3B1 block = "begin " labeled $('; labeled) "end" ;

      9B3B2 branch = bruto / brxto;

         9B3B2A brxto = "brx " topart :cbrxto[2];

         9B3B2B bruto =

         9B3B2B1 (("bru " / "go ") topart /

         9B3B2B2 "goto " adrarg) :cbruto[2];

         9B3B2C topart = "to" adrarg;

         9B3B2D  adrarg  =  (exp  actual $(', exp actual :do[2]) /
         .empty);

      9B3B3 suber = call /return / entry;   /cojump

         9B3B3A return =

         9B3B3A1 "return " actual :crtn[1]/

         9B3B3A2 ("brr " topart :cbrr/

         9B3B3A3 "sbrr " topart :csbrr)[2];
```

```
9B3B3B call =

9B3B3B1 ("call "  adrarg :ccall/

9B3B3B2 "brm " topart :cbrm/

9B3B3B3 "sbrm " topart :csbrm)[2];

9B3B3C entry = "entry " .id formal :centry[2];

9B3B4 iterat = for / while / over ;

9B3B4A for =  "for " .id "from " exp ("inc " :inc /"dec "
:dec)[0] exp "to " exp "do " :cfor1[4] * stat :cfor2[0];

9B3B4B while = "while  "  exp "do" #1 :whil1[2] * stat #1
:whil2[1];

9B3B4C over = "over " .id '[  .id  (.num  /.empty)  '] do
stat;

9B3B5 case = ithcse / test;

9B3B5A ithcse = "case " exp "of"  "begin" stat $('; stat)
"end" ;

9B3B5B  test = "test " exp "of" begin casest $('; casest)
("otherwise" stat /.empty :mt[]) ;

9B3B5B1 casest = (binrel / exp) ': stat;

9B3B6 other = bump /null /exu /arpas /copy ;

9B3B6A bump = "bump " (

9B3B6A1 "down" adrlst :bmpdwn[1] /

9B3B6A2 ("up" /.empty) adrlst :bmpup[1]);

9B3B6A3 adrlst = exp $(', exp :do[2]);

9B3B6B null = "null"  :mt[0];

9B3B6C exu = "execute " exp :exu[1];

9B3B6D copy = "copy " ;

9B3B6E cpybit  =  'a  / 'b / 'x / 'e / 'n / "ab" / "ax" /
"ba" / "bx" / "xa" / "xb";
```

```
9B4I exp =  bexp
   9B4A "<->" bexp :xchang[2] /
   9B4B $(', bexp :do[2])
      9B4B1 ('+
         9B4B1A ('+  exp :addmem[2] /
         9B4B1B .empty exp :store[2]);
   9B4C bexp  =  "if "  union "then " :iftest[1] * union ("else "
   exp) / union;
   9B4D union = inter ("or "  union :or[2] /.empty);
   9B4E inter = neg ("and "  inter :and[2] /.empty);
   9B4F neg = "not " negneg / relat ;
   9B4G negneg = "not " neg / relat :not[1] ;
   9B4H relat =
      9B4H1 ".pos" addr :pos[1]/
      9B4H2 ".neg" addr :neg[1] /
      9B4H3 ".skip" prim :skip[1] /
      9B4H4 ".decpos" prim :decpos[1]/
      9B4H5 ".decneg" prim :decneg[1]/
      9B4H6  sum ( binrel / .empty);
   9B4I binrel =
      9B4I1 ".lt" sum :lt[2] /
      9B4I2 ".le" sum :le[2] /
      9B4I3 ".eq" sum ('& sum :msk[3] / .empty) :eq[2] /
      9B4I4 '& sum :msk[2] (
         9B4I4A ".eq" sum :eq[2] /
         9B4I4B ".ne" sum :ne[2]) /
```

```
        9B4I5 ".ne" sum ('& sum :msk[3] / .empty) :ne[2] /

        9B4I6 ".ge" sum :ge[2] /

        9B4I7 ".gt" sum :gt[2] /

        9B4I8 ".cb " sum :cb[2] /

        9B4I9 ".ncb " sum :ncb[2] /

        9B4I10 ".(" sum ', sum '] :int[3] ;

     9B4J sum = prod (('+ sum :add / '- sum :sub)[2] /.empty);

     9B4K prod = factor (('* prod :mult / '/ prod :divid / '↑ prod
     :rem)[2] /.empty);

     9B4L factor = bor / '- factor :minus[1];

     9B4M bor = band ((".v " bor :mrg/ ".x " bor :eor)[2] /.empty;

     9B4N band = prim (".a " band :etr[2] /.empty);

     9B4O prim = bltin / abxreg / varfun / const / immed / indir /
     '( exp ') / arpas ;

       9B4O1 abxreg = (".ar" :areg / ".xr" :xreg / ".br" :breg)[0];

       9B4O2 bltin = shift / brs ;

          9B4O2A shift = shiftl actual (",2" :tagged[0] /.empty
          :mt[0]) :cshift[3];

          9B4O2B shiftl = (".lrsh" :lrsh /".lsh" :lsh /".rsh" :rsh
          /".rcy" :rcy/ ".lcy" :lcy)[0];

          9B4O2C brs = ".brs" sinum actual :cbrs[2];

       9B4O3 varfun = .id ('( index ') /.empty) actual ;

  9B5 addr = var / indir / immed / const ;

    9B5A sinum = ('- const :scon[1] / const ) ;

    9B5B index = '[ sum '] :indx[1] ;

    9B5C immed = '$ (var / sinum ( index / .empty) / .sr)
    :cimmed[1] ;

    9B5D indir = '[ (immed / var / const) '] :cindir[1] ;
```

```
        9B5E const = .num :con[1] ;

        9B5F var = .id (index /.empty mt[0]) :cvar[2];

    9B6 actual = '( act1 act2 act2 ') :act[3] / .empty :mt[] ;

        9B6A act1 = exp / .empty :mt[0] ;

        9B6B  act2 = ', / .empty :mt[0];

  9C %unparse rules%

    9C1 % declarations %

        9C1A  cext[do[-,-]] => cext[*1:*1] cext[*1:*2]

            9C1A1  [-]   =>   *1   "   ext"\  (.ta[*1,alcted] ?error  /
            .sa[*1,extnr1]);

        9C1B cequ[do[-,-]] => cequ[*1:*1] cequ*]:*2]

            9C1B1 [-] => *1:*1 " eu "*2\;

    9C2 % basic  executable statements%

        9C2A %while%

            9C2A1 whil1[-,#1,#2] => def[*2] whilx[*1,#2];

            9C2A2 whilx[-,#2] => lopr[*1,#1,#2] brf[*1,#2] def[*2];

            9C2A3 whil2[#1,#2] =>  bru[#1] def[*2];

        9C2B %if%

            9C2B1 if1[-,#2] => lopr[*1,#1,*2] brf[*1,#2] def[#1];

            9C2B2 if2[#1,#2] => bru[#2] def[#1];

        9C2C %branch%

            9C2C1 bru[#1] => "bru " #1\;

            9C2C2 cbruto[-,-] => cgoto["bru",*1,*2];

            9C2C3 cbrxto[-,-] => cgoto["brx",*1,*2];

            9C2C4 cbrrto[-,-] => cgoto["brr",*1,*2];

            9C2C5 csbrrto[-,-] => cgoto["sbrr",*1,*2];
```

```
            9C2C6 cbrmto[-,-] => cgoto["brm",*1,*2];

            9C2C7 csbrm[-,-] => cgoto["sbrm",*1,*2];

            9C2C8 ccall[-,-] =>

               9C2C8A .tf rentrt cgoto["sbrm",*1,*2] /

               9C2C8B cgoto["brm",*1,*2];

            9C2C9 crtn[-] =>

               9C2C9A .tf popprc "brr 0"\ /

               9C2C9B .tf rentrt cgoto["sbrm",?,*1]

               9C2C9C cgoto["brr",?,*2];

            9C2C10 cgoto[-,-,-] =>

               9C2C10A [*3 token[*1] oper[*1,*1] /

               9C2C10B work[*2] "bru* " .w\;

        9C2D cexu[-] => oper["exu",*1];

        9C2E  centry[-,-]  =>  bru[#1]  '$ *1 " zro " ?procname *2 (.tf
        rentrt / .empty ??) def[#1];

     9C3 % instructions with an addess field %

        9C3A oper[-,cindir[-]] => oper1[*1,*2,'*]

           9C3A1 [-,-] => oper1[*1,*2,' ];

        9C3B oper1

           9C3B1 [-,cvar[-,mt[],-] => '  *1 *3 *2:*1\

           9C3B2 [-,cvar[-,-],-] => prendx[*2:*2]  '   *1  *3  '  *2:*1
           ",2"\

           9C3B3 [-,cimmed[.id,mt[]],-] => ' *1 *3 '  '= *2:*1\

           9C3B4  [-,cimmed[.id,-],-]  => prendx[*2:*2] '  *1 *3 '  '=
           *2:*1 ",2"\

           9C3B5 [-,cimmed[con[],mt[]]] => '  *1 *3  *2:*1:*1\

           9C3B6 [-,cimmed[scon[-],mt[]]] => ' *1 *3 '- *2:*1:*1\
```

```
     9C3B7  [-,cimmed[con[],-]]   =>   prendx[*2:*2] "   *1  *3
     *2:*1:*1 ",2"\

     9C3B8  [-,cimmed[scon[-],-]] => prendx[*2:*2] "  *1  *3 "-
     *2:*1:*1 ",2"\

 :9C3C prendx[mt[]] => .empty

     9C3C1 [add[con[-],-]] => loadx[*1:*2] " eax " *1:*1 ",2"\

     9C3C2 [add[-,con[-]]] => loadx[*1:*1] " eax " *2:*1 ",2"\

     9C3C3 [sub[con[-],-]] => loadx[*1:*2] " eax -" *1:*1 ",2"\

     9C3C4 [sub[-,con[-]]] => loadx[*1:*1] " eax -" *2:*1 ",2"\

     9C3C5 [-] => loadx[*1];

 :9C3D loadx[xreg[]] => .empty

     9C3D1 [areg[]] => " cax "\

     9C3D2 [-] => token [*1] oper["ldx",*1] /( tryb[*1] " cbx"\ /
     loada[*1] " cax"\);

 :9C3E token[cvan[-,mt[]]] => .empty

     9C3E1 [cvar[-,-]] => xtoken[*1:*2]

     9C3E2 [con[-]] => .empty

     9C3E3 [cindir[-]] => token[*1:*1]

     9C3E4 [cimmed[-]] => token[*1:*1]

     9C3E5 [scon[]] => .empty

     9C3E6 [.id] => .empty

     9C3E7 [.num] => .empty

 :9C3F xtoken[cvar[-,-]] => xtoken[*1:*2]

     9C3F1 [add[-,con[]]] => xtoken[*1:*1]

     9C3F2 [add[con[],-]] => xtoken[*1:*2]

     9C3F3 [sub[-,-]] => xtoken[add[*1:*1,*1:*2]]

     9C3F4 [xreg[]] => .empty
```

```
        9C3F5 [areg[]] => .empty

        9C3F6 [breg[]] => .empty;

    9C4 % logical operation and branches%

      9C4A   lopr[or[-,-],#1,-]  =>  lopr[*1:*1,#1,#2]  brt[*1:*1,#1]
      def[#2] lopr[*1:*2,#1,*3]

        9C4A1  [and[-,-],-,#1]  =>  lor[*1:*1,#2,#1]  brf[*1:*1,#1]
        def[#2] lopr[*1:*2,*2,#1]

        9C4A2 [not[-],#1,#2] => lopr[*1:*1,#2,#1]

        9C4A3 [-,-,-] => .empty;

      9C4B brt[or[-,-],#1] => brt[*1:*2,#1]

        9C4B1 [and[-,-],#1] => brt[*1:*2,#1]

        9C4B2 [not[-],#1] => brf[*1:*1,#1]

        9C4B3 [le[-,-],#1] => ble[*1:*1,*1:*2,#1]

        9C4B4 [lt[-,-],#1] => blt[*1:*1,*1:*2,#1]

        9C4B5 [eq[-,-],#1] => beq[*1:*1,*1:*2,#1]

        9C4B6 [ge[-,-],#1] => bge[*1:*1,*1:*2,#1]

        9C4B7 [gt[-,-],#1] => ble[*1:*2,*1:*1,#1]

        9C4B8 [ne[-,-],#1] => bne[*1:*1,*1:*2,#1]

        9C4B9 [pos[-],#1] => bpos[*1:*1,#1]

        9C4B10 [neg[-,],#1 => bneg[*1:*1,#1]

        9C4B11 [cb[-,-],#1] => bcb[*1:*1,*1:*2,#1]

        9C4B12 [ncb[-,-],#1] => bncb[*1:*1,*1:*2,#1]

        9C4B13 [int[-,-,-],#1] => bint[*1:*1,*1:*2,*1:*3,#1]

        9C4B14 [-,#1] => loada[*1] " ske =0;" bru[*2];

      9C4C brf[or[-,-],#1] => brf[*1:*2,#1]

        9C4C1 [and[-,-],#1] => brf[*1:*2,#1]
```

```
9C4C2 [not[-],#1] => brt[*1:*1,#1]

9C4C3 [le[-,-],#1] => ble[*1:*1,*1:*1,#1]

9C4C4 [lt[-,-],#1] => bge[*1:*1,*1:*2,#1]

9C4C5 [eq[-,-],#1] => bne[*1:*1,*1:*2,#1]

9C4C6 [ge[-,-],#1] => blt[*1:*1,*1:*2,#1]

9C4C7 [gt[-,-],#1] => ble[*1:*1,*1:*2,#1]

9C4C8 [ne[-,-],#1] => beq[*1:*1,*1:*2,#1]

9C4C9 [pos[-],#1] => bneg[*1:*1,#1]

9C4C10 [neg[-],#1] => bpos[*1:*1,#1]

9C4C11 [cb[-,-],#1] => bncb[*1:*1,*1:*2,#1]

9C4C12 [ncb[-,-],#1] => bcn[*1:*1,*1:*2,#1]

9C4C13 [int[-,-,-],#1] => bintf[*1:*1,*1:*2,*1:*3,#1]

9C4C14  [-,#1] => (tryb[*1] " skb =-1; bru *+2;" / loada[*1]
" ske =0; bru *+2;") bru[#1];

9C4D  blt[-,-,#1] => (

9C4D1 token[*1] loada[*2] oper["ske",*1] oper["skg",*1] /

9C4D2 work[*1] loada[*2]  wrk1["ske"] wrk2["skg"]  " bru
*+2;" ) bru[*1];

9C4E  ble[-,-,#1] => (

9C4E1 token[*2] loada[*1] oper["skg",*2] /

9C4E2 token[*1] loada[*2] oper["skg",*1] " bru *+2"\ /

9C4E3 work[*2] loada[*1] wrk2["skg"]) bru[#1];

9C4F  beq[-,-,#1] => (

9C4F1 token[*2] loada[*1] oper["ske",*2] /

9C4F2 token[*1] loada[*2] oper["ske",*1] /

9C4F3 work[*2] loada[*1] wrk2["ske"]) " bru *+2;" bru[#1];
```

```
9C4G  bge[-,-,#1] => (
    9C4G1 token[*1] loada[*2] oper["ske",*1] oper["skg",*1] /
    9C4G2 work[*1] loada[*2] wrk1["ske"] wrk2["skg"]) bru[#1];
9C4H  bne[-,-,#1] => (
    9C4H1 token[*2] loada[*1] oper[*2] /
    9C4H2 token[*1] loada[*2] oper["ske",*1] /
    9C4H3 work[*2] loada[*1] wrk2["ske"]) bru[*1];
9C4I  bpos[-,#1] =>
    9C4I1 token[*1] oper["skn",*1]  bru[#1]  /
    9C4I2 (tryb[*1] " skb =40000000b; bru *+2;" /
    9C4I3 loada[*1] " ska =40000000b; bru *+2;") bru[#1];
9C4J  bneg[-,#1] =>
    9C4J1 token[*1] oper["skn",*1] " bru *+2;" bru[#1] /
    9C4J2 (tryb[*1] "skb =40000000b;" /
    9C4J3 loada[*1] " ska =40000000b;") bru[#1];
9C4K bcb[-,-,#1] => (
    9C4K1 token[*1] (
        9C4K1A tryb[*2] oper["skb",*1] /
        9C4K1B loada[*2] oper["ska",*1]) /
    9C4K2 token[*2] (
        9C4K2A tryb[*1] oper["skb",*2] /
        9C4K2B load[*1] oper["ska".*2] /
    9C4K3 work[*1] loada[*2] wrk2["ska"]) bru[#1];
9C4L bncb[-,-,#1] => (
    9C4L1 token[*1] (
```

```
            9C4L1A tryb[*2] oper["skb",*1] /
            9C4L1B loada[*2] oper["ska",*1]) /
         9C4L2 token[*2] (
            9C4L2A tryb[*1] oper["skb",*2] /
            9C4L2B load[*1] oper["ska".*2] /
         9C4L3  work[*1] loada[*2] wrk2["ska"]) " bru *+2;" bru[#1];
      9C4M bint[-,-,-,#1]  => skg skg bru *+2 bru true
      9C4N bintf[-,-,-,#1]  => skg skg bru false
   9C5 %expression evaluation%
      9C5A comop[-,-,-] =>
         9C5A1 token[*2] loada[*1] oper[*3,*1] /
         9C5A2 token[*1] loada[*2] oper[*3,*1] /
         9C5A3 work[*1] loada[*2] wrk2[*3];
      9C5B add[minus[-],-] => sub[*2,*1:*1]
         9C5B1 [-,-] => comop[*1,*2,"add"]
      9C5C mrg[-,-] => comop[*1,*2,"mrg"];
      9C5D etr[-,-] => comop[*1,*2,"etr];
      9C5E eor[-,-] => comop[*1,*2,"eor"];
      9C5F sub[-,-] =>
         9C5F1 token[*2] loada[*1] oper["sub",*2] /
         9C5F2 token[*1] (                      -- not needed
            9C5F2A tryb[*2] " cba; cna;" /
            9C5F2B loada[*2] " cna;") oper["add",*1] /
         9C5F3 work[*2] loada[*1] wrk2["sub"];
      9C5G minus[con[-]] => (
```

```
    9C5G1 token[*1] oper["lda",*1] /
    9C5G2 tryb[*1] " cba;" /
    9C5G3 loada[*1]) " cna"\;
9C5H divid[-,-] =>
    9C5H1 token[*2] loada[*1] " rsh 23;" oper[*2] /
    9C5H2 work[*2] loada[*1] " rsh 23;" wrk2["div"];
9C5I   tryb[mult[-,-]] =>
    9C5I1 (
        9C5I1A token[*1:*2] loada[*1:*1] oper["mul",*1:*2] /
        9C5I1B token[*1:*1] loada[*1:*2] oper["mul",*1:*1] /
        9C5I1C work[*1:*1] loada[*1:*2] wrk2["mul"]) " rsh 1"\
    9C5I2 [rem[-,-]] => divid[*1:*1,*1:*2]
    9C5I3 [breg[]] => .empty;
9C5J mult / => err;
9C5K rem / => err;
9C5L and / => err;
9C5M or / => err;
9C5N not/ => err;
9C5O pos / => err;
9C5P neg / => err;
9C5Q skip / => er;
9C5R lt / => err;
9C5S le / => err;
9C5T eq / => err;
9C5U ne / => err;
```

```
    9C5V ge / => err;
    9C5W gt / => err;
    9C5X cb / => err
    9C5Y ncb/ => err;
  9C6 % a an b register loading%
    9C6A loada[areg[] => .empty
      9C6A1 [xreg[]] => "cxa"
      9C6A2  [-] => (
        9C6A2A token[*1] oper["lda",*1] /
        9C6A2B tryb[*1] "cba"\ / *1;
    9C6B loadb[areg[]] => "cba"
      9C6B1 [xreg[]] => "cxb"
      9C6B2 [[breg[]] => .empty
      9C6B3 [-] => (
        9C6B3A token[*1] oper["ldb",*1] /
        9C6B3B tryb[*1] /
        9C6B3C loada[*1] "cba"\);
```

```
10 .HED=" MOL - SUPPORT LIBRARY"; .PGN=.PGN-1; .RES;
```

II .HED=" MOL - BIBLIOGRAPHY"; .PGN=.PGN-1; .RES;

MOL - BIBLIOGRAPHY

IIA (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.

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

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

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

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

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

IIG (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.

(Andrews1) TreeMeta

/REWRI, 09/26/68 0851:18 JFR ;

1 Introduction

1A The Tree Meta, MOL, and SPL compilers are in need of many changes. This memo discusses the problems with the compilers in their current stite and offers a unified solution.

1B We fell that a total planned rewrite of all three compilers offers the most economical long term solution. Eventually all the things listed below must be done. If they can be accompolshed with some simultianenty, all changes can be accomodated on the first pass, less time will be wasted, and the benifits of the rewrite will be available sooner.

2 Current Problems and Proposed Solutions

2A MOL bugs

2A1 The most straight forward problems are the bugs in the MOL. None of these are too serious for the current MOL use. We all just avoid using the syntatic phrases that cause the problems. This does mean, however, that the code we write does not always reflect the original, notural conception. The rarity of this certaintly does not warrent a complete rewrite of the MOL, Most of the bugs could be fixed by a couple of weeks work on the current version.

2A2 A more serious problem with the MOL is the 80 character line orentitation of the input routines. These programs rely on the format of QED lines, thus code that exists in NLS format must be made to look like QED format before compilation. This limits the length of NLS statements containing MOL code, and just make everything kluedgy

2B Symbol problem

2B1 NLS has grow to such porpotions that it nearly overflows the symbol tables of the TSS subsystems used to assemble, load, and debug it. Already it is too large to use NARP and DDT, we must use ARPAS and CDDT.

2B2 If an additive assembler were added to TM, and MOL were rewritten using the builtin assembler, this problem would completely disappear. The additive assembler would avoid the symbolic definition of the many thousand generated symbols that MOL currently produces. The only symbols defined at load time would be those specifically defined in the MOL code. This would reduce the total number from about 3000 to a few hundred.

2B3 WE would design the additive assembler so that the files it produces would be NARP-DDT compatiable. This would mean that we could use the new DDT and its improved debugging features.

2C SYstem load

2C1 The loac that assembling NLS currently puts on the TSS is detrimental in two ways. It kills the response of the system, eats up a lot of RAD space and makes NLS close to unusable while it is beign done. This, in turn, makes the system programmers a little afraid to do assemblies and thus slows system design and debugging. This last problem is felt is a slow creeping way every time we put off doing an assembly for a few days because they put such a load on the system.

2C2 The present way of assembling the system is to first compile all the files, then assemble them, an finally load them. It takes longer to assemble a file than it does to compile it, thus getting rid of the assembly phase would cut the process in half. Moreover, the compilers currently spend more than half their time in the symbolic output phases, cutting the time again in half. and finally the symbol table routine waste coniderable time in long compilations. We partially implemented a hash table in the MOL and compilation time droped by 1/4 for large compilations. All this means that total assembly time would drop by a factor of 5 anc maybe even 10.

2C3 The major effort for the conversion to additive assemblers would be done once, in TM. The syntax for additive assembly output would closely resemble the current syntax for symbolic output.

2D Coherent package with NLS

2D1 A minor but anoying feature of the compilers as they currently stand is their kluedgy interface with NLS. This is espically true when it comes to error recovery. While everyting else is being rewirtten be could devise a general scheme for file processing and feedback to the user about the results of the process.

2D2 If the MOL and the SPLs were both written in TM, the code files for the system could be better organized. Each overlay could be a single file, the binary would be the result of a single compilation.

2D2A This would simplify system assembly as well as speed it up. Less RAD space would be needed because fewer intermediate files would be generated. Fewer symbolic and binary files would have to be saved on thm disc.

2D2B Also, by having the files more closely related to program function better usage could be made of the NLS linkage commands.

2E Powerful syntax in MOL

2E1 A number of new features will be added to the MOL syntax. These are discussed in more deatil below under MOL. The main benifit of the features is that they will make the syntax of the language closer to the intentions of the codder. This does not change anything in drastic ways, it just makes life a little better when someone is trying to figure out what a piece of code is "supposed" to do.

2F More Dense SPL code

2F1 by rewriting the SPLs and using the features of TM, we feel that about a 20% reduction could be made in the amount of instructions compiled. This does not affect NLS is a big way, but it would gives us a little more room for expansion in some of the overlay pages that are currently over 90% full.

2G Pratice what we preach

2G1 Converting the code files to NLS, and retaining the current compiler systems is doing only half a job. The listings would not disappear, and the "larger NLS experiment" will not be done. To replace the listings the code files must be coherently organized and easily accessiable. For files written in MOL this may mean expermienting with syntatic changes, and this is only pratical if they are written in TM.

2G2 Eventually we would like to work out a method of compilation that substituted the tree structure on NLS files for the phrase tructure of the MOL and SPL. This is virtually impossible unless the MOL is in TM and the changes can be done in one central place, namely the TM library, for all the expreimental compiles.

2G3 There is the vague illusive notion of staying on top of the design problem. The code files are becomming cumbersome to work with in their current form. Just moving them to NLS would not help much. If however, the syntax of the languages were more suited to NLS linkage conventions, and the files themselves were better structured we may again reach a point of feeling that the structure is well understood, and the effect of changes in code can be properly perdicted.

2G4 We finally have figured out a way of writing the parse and unparse rules for the MOL compiler in TM and not overflowing the push down stacks during compilation. Now that we have a solution it would be satisfying to get everything

3 Proposed changes

3A Tree Meta

3A1 Additive Assembler

3A1A This is one of the major projects in terms of radical changes to the existing TM system. TM would be enlarged to permit either symbolic or binary output from a compilation. The binary output would be formed by making up words for a sort of backhalf processor that puts the words in the percise form necessary for DDT. Linkage for undefined labels and packing of undefined polish expressions would be automatically handeled by the backhalf.

3A2 Symbol Table

3A2A The new symbol table will use hash entry instead of the current search technique. In conjunction with the additive assembler, it will be expanded to include declaration flags, array size parameters, and definition bits.

3A2B The new table would also reserve bits for compile time attribute flags. This would permit a TM compiler to check declarations and give appropiate diagnostics.

3A3 Basic recognizers

3A3A The basic recognizers will be changed to delete blanks after recognition instead of before. This will reduce the initial recognizer test, and thus the time for a failure, to less than 5 from the current 25. These failures represent about 20% of the runtime for a compilation.

3A3B The "TST" (literal string test) recognizer will be further improved so that a failure will average only silghtly more tham 3 instructions. This recognizer represents about 80% of the total recognizers executed. Moreover its failure to success ratio is about 20 to 1.

3A4 Use of Skip return

3A4A A new convention will be established for all the recognizers and recursive rules. The return will skip if the subroutine was successful and not skip if it failed. This means that the current branch false instruction can be done away with. It is the shortest and yet most frequently executed pop in the TM system. It accounts for about 35% of the pops executed.

3A5 interface to NLS

3A5A Once TM has been interfaced to NLS, all the other compilers should interface automatically. It is hard to guess long it will take to do the job for we do not yet know what we want to do.

3A5B One suggession is to add to NLS the ability to store a list of t-pointers, which are the result of a compilation. This list could be kept be NLS with the file until another process is preformed on the file. The statements on the list would be displayed under a new view-spec parameter.

3B MOL

3B1 Rewrite in Tree Meta

3B1A The entire MOL will be written in the new TM language using the additive assembler. This project is mostly done. We have a version of the MOL written in an extended TM language using symbolic output. The code is almost complete and we do not anticipate any new problems. Of corse the compiler cannot be checked out without a new TM because it needs features in the metalanguage not currently in TM.

3B2 New features

3B2A The new Mol will have many additional features. None of them are expensive in terms of effort or compile time. They come mostly for free with the use of TM for compiling.

3B2B The new compiler will allow an expression to be a statement. This will help by clearing up the meaning of many lines of current code, when an expression is forced into an assign statement even though that is not the intention of the writer.

3B2C The store opeator, currently available only through the assign statement, will be put in as the lowest level binding operator in an expression. This will mean that stores can be done during expression evaluation. This also helps conciseness and clarity.

3B2D Possible addresses will be expanded from the currently restirced set to any expression. This was always wanted, even in the original MOL specification, but was too difficult to add to the original version. The power of TM to do its top-down tree search means that the more versitle syntax can be added and tight code can still be produced for the simple cases, just as it is now.

3B2E The double branch currently compiled at the end of logical expressions will disappear. This can be simply with the unparse rules in TM, it would have been difficult with the current MOL.

3B2F We plan to introduce a new case sttement. It will do a single case based on a logical expression at the start of the case rather than a predetermined number.

3B2G Syntax will be added to simplify the use of the brx, skr, xma, and the register exchange instructions. This will make all the 940 instruction available directly in the MOL except those concerned with floating point exponents.

3B3 Use of Addiive Assembler

3B3A When the MOL is written using the additive assembler all the many generated labels will just not apear in the binary file. This will mean that the number of symbols for NLS will reduce to a mangeable size. Moreover, our current kludegy way of using the frozen feature of ARPAS can given up completely.

3B4 complete integration into MOLR

3B4A The already existing version of MOL in TM is in the MOL report file, which is in NLS format on the disc. This file

represents the first attempt to integrate the actual code for a compiler into the formal and informal description. This integration is only possible because the TM code for MOL is

breif enough to fit in a file with the report. It may well be that this file (MOLR) could be the first realistic attempt

3B5 Transfer of current code to NLS

3B5A We already have a program, PASSO, which reads an MOL program from a QED file and produces another QED file in structured statement form. The structure is determined by a set of rules for indenting close to the set used by Mckeeman in his uncrunch program (cacm 65). We have used this program in conjunction with the insert QED branch NLS command with complete success, and feel that the initial transfer should be a straight fordward taks of only a few days.

3C SPL

3C1 Use of Addiive Assembler

3C1A When the SPL compiler and the MOL compiler are in TM, then can be rigged to output to a continuous file. This will mean tha a single NLS file can contain code in both languages and still be compiled in one simple operation.

3C2 Clarity of code in SPLs

3C2A If the SPL compiler is in TM the parse rules will contain only parse information and node building cirections. This should make them much more readable, a feature always wanted by those that try to figure out commands of NLS by reading the code in the SPLs.

3C3 A report on the SPL is about 3/4 done (currently about 50 pages). When the SPL compiler is rewritten the new version would be integrated into the report. This would be another large scale attempt to do away with listing by organizing the documenation and code into an easily accessiable monolithic structured NLS fil.

4 Manpower estimates

4A To reap the full benifits from these changes, all the projects must be done as a whole. MOL and SPL cannot be rewritted without rewritin TM. And it does little good to only rewrite TM. Thus, although the esitmeates are broken down, the entire project must be compleleted to be worth the effort.

4B The estimate to rewrite TM, and bring the report up to publishable standarcs is 2 mar-months. The report, much as it appeard in the ROme report on the disc as a single NLS file. The new TM library and compiler will be a part of the , and the report will be kept in sync with the new compiler. Most of the 2 months will be devoted to the new library and the aditive assembler.

4C after the new TM is done the MOL sould only take about 1.5 more man-months. This is again for finishing the new TM version of the compiler and brining the report file up to date and in publishible form.

4D Rewriting the SPL is the simplist of the tasks. We estimate one man-month to both redo the compiler, and finish the SPL report. About 1/3 of the time will be spent on the compiler and about 2/3 on the report.

4E These estimates are made in terms of time spent doing the work. Normally within the center, the programmers spend a good deal of their time debugging NLS, working on specifications and ideas for new features, and gererally doing small detailed tasks not realted to a specific project. With this in mind, it becomes very difficult to estimate the real time these projects will require.

5 Pass4

Store (returns in .AR) (BF)

Pack

< LDP STRP > — end of string storage area

if .SKIP .BRS5(5, HTT) then return ( )

else begin

if .NEG HTT[2] then call SERR(8)

else begin (.BRS36( ))

SSP[0]←SSP[1] end.

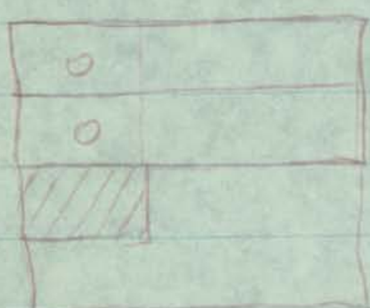
HTT ZRO HT

ZRO EAT

ZRO 0 ← initialize to zero

& set HT to zero

STRP



char before 1st char MXSTR

last character

8c characters

STR

end of string storage.

word addr × 3 + (0, 1, 2)

CIC LDP STRP

WCI* IWP

gci wci

write on end

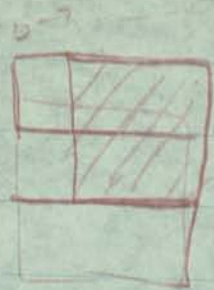
get

B → D HT

A val

SKIP [illegible]

[illegible]



GCL & WXI

GHT

BRS 6 new entry

5 (A, B) . # table [illegible]

gcl

for tree (HT)

& String pops

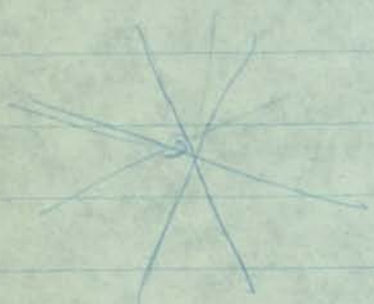
Mol ABS, MIN, MAX

3

5th loses soft edges
in averaging

does not make
edges

t a c



cat

t c

$

$(X + N/X$

row

ck

120

— I, J

{ 0 origin
character or not

array [I] → .XR

J/3

.XR

*36

.AR ← array [(I + J/3)]

Roberts 3 adds sub
add
2 mpy mpy

(6) 1 root (4) root

Prewitt
2 sub sub
1 add mpy add mpy
(3) (2)

Sobels 13 adds 4 adds (4 shifts)
a 1 bit shifts 1 mpy
1 root
(29) 2 mpy (6) 1 root

LOC 4 adds 2 adds

3 compares

(8) 1 shift

3/A 4
6 adds

backup or

non backup: mode

.DELIMITER (no gobble)

.BLANKS

$ID

.ID .BLANKS

STP[0] ← SSP[0]

"FOR"

"FOR"(s-s)

= #

got it
STP[1] ← SSP[0]
store, gobble blanks, set flag

failure SSP[0] ← STP[0] {reset SS}
reset window
reset flag

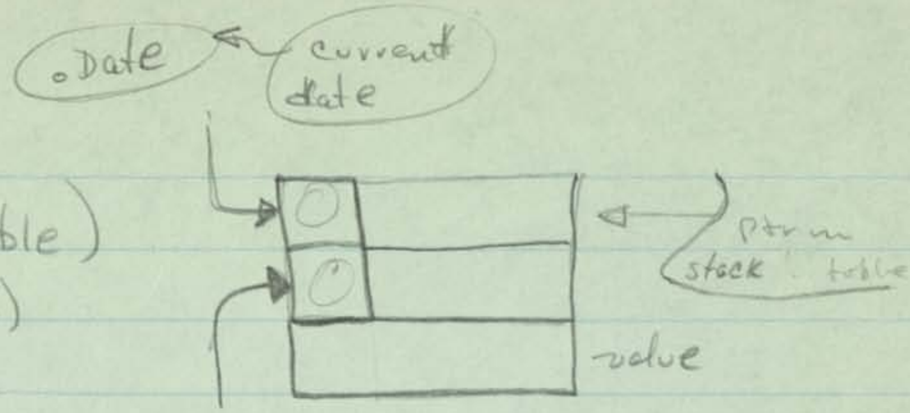
TST 6 instructions

SKSE {literal string}

lda = 1st character
SKE + IWP
no BRU failure

read length of TST into window
use SKSE
no
yes

maybe TST
no
yes



String (double)
Character (single)

16 ~~attribute flags~~ 0

~~BRS~~ OUTS

```
CAX
LDP   HT,2
LDX   LITF
BRS   35
```

```
cax
LCHNO ← + (HT+1,2) − (HT,2)
  .BRS35( HT+2, HT+1,2, LITF)
```

```
TCHNØ ← +  HT[.XR] − HT[.XR+1]
RETURN ( .BRS35( HT[.XR], HT[.XR+1], TELNO))
```

ØUTN

```
BRS 36 (.AR , −10, FNUMØ)
```

# of digits ?

```
LDA   LDX

SKG
BRX   *-1
CXA
CNA
ADM
```

if .NEG then

}

2

$\log_{10} = \frac{\log_2}{\log_{10} 2}$ 100   6   neg ?

.3   1.2   →

$\log_{10}$ [1000] = 3 $\log_2$ $\frac{8}{3}$ 2.4 → 3 copy AB, BX; \EXT14

I+15

gen temps

4

~~Prefix~~ Temp locations = '___' — symbol table

STB ~~HW~~

Bumps add w
add v TempTab[w] to [ ]
putc LC to
set max w

Declare

| A[I+3]
[A+3] ,I

Common temps

00BREAK00

00BREAK00

META II

A SYNTAX-ORIENTED COMPILER WRITING LANGUAGE

D. V. Schorre
UCLA Computing Facility

META II is a compiler writing language which consists of syntax equations resembling Backus normal form and into which instructions to output assembly language commands are inserted. Compilers have been written in this language for VALGOL I and VALGOL II. The former is a simple algebraic language designed for the purpose of illustrating META II. The latter contains a fairly large subset of ALGOL 60.

The method of writing compilers which is given in detail in the paper may be explained briefly as follows. Each syntax equation is translated into a recursive subroutine which tests the input string for a particular phrase structure, and deletes it if found. Backup is avoided by the extensive use of factoring in the syntax equations. For each source language, an interpreter is written and programs are compiled into that interpretive language.

META II is not intended as a standard language which everyone will use to write compilers. Rather, it is an example of a simple working language which can give one a good start in designing a compiler-writing compiler suited to his own needs. Indeed, the META II compiler is written in its own language, thus lending itself to modification.

History

The basic ideas behind META II were described in a series of three papers by Schmidt,[1] Metcalf,[2] and Schorre.[3] These papers were presented at the 1963 National A.C.M. Convention in Denver, and represented the activity of the Working Group on Syntax-Directed Compilers of the Los Angeles SIGPLAN. The methods used by that group are similar to those of Glennie and Conway, but differ in one important respect. Both of these researchers expressed syntax in the form of diagrams, which they subsequently coded for use on a computer. In the case of META II, the syntax is input to the computer in a notation resembling Backus normal form. The method of syntax analysis discussed in this paper is entirely different from the one used by Irons[6] and Bastian.[7] All of these methods can be traced back to the mathematical study of natural languages, as described by Chomsky.[8]

Syntax Notation

The notation used here is similar to the meta language of the ALGOL 60 report. Probably the main difference is that this notation can be keypunched. Symbols in the target language are represented as strings of characters, surrounded by quotes. Metalinguistic variables have the same form as identifiers in ALGOL, viz., a letter followed by a sequence of letters or digits.

Items are written consecutively to indicate concatenation and separated by a slash to indicate alternation. Each equation ends with a semicolon which, due to keypunch limitations, is represented by a period followed by a comma. An example of a syntax equation is:

```
LOGICALVALUE = '.TRUE' / '.FALSE' .,
```

In the versions of ALGOL described in this paper the symbols which are usually printed in boldface type will begin with periods, for example:

```
.PROCEDURE .TRUE .IF
```

To indicate that a syntactic element is optional, it may be put in alternation with the word .EMPTY. For example:

```
SUBSECONDARY = '*' PRIMARY / .EMPTY .,
SECONDARY = PRIMARY SUBSECONDARY .,
```

By factoring, these two equations can be written as a single equation.

```
SECONDARY = PRIMARY('*' PRIMARY / .EMPTY) .,
```

Built into the META II language is the ability to recognize three basic symbols which are:

1. Identifiers -- represented by .ID,
2. Strings -- represented by .STRING,
3. Numbers -- represented by .NUMBER.

The definition of identifier is the same in META II as in ALGOL, viz., a letter followed by a sequence of letters or digits. The definition of a string is changed because of the limited character set available on the usual keypunch. In ALGOL, strings are surrounded by opening and closing quotation marks, making it possible to have quotes within a string. The single quotation mark on the keypunch is unique, imposing the restriction that a string in quotes can contain no other quotation marks.

The definition of number has been radically changed. The reason for this is to cut down on the space required by the machine subroutine which recognizes numbers. A number is considered to be a string of digits which may include imbedded periods, but may not begin or end with a period; moreover, periods may not be adjacent. The use of the subscript 10 has been eliminated.

Now we have enough of the syntax defining features of the META II language so that we can consider a simple example in some detail.

The example given here is a set of four syntax equations for defining a very limited class of algebraic expressions. The two operators, addition and multiplication, will be represented by + and * respectively. Multiplication takes precedence over addition; otherwise precedence is indicated by parentheses. Some examples are:

```
A
A + B
A + B * C
(A + B) * C
```

The syntax equations which define this class of expressions are as follows:

```
EX3 = .ID / '(' EX1 ')' .,
EX2 = EX3 ('*' EX2 / .EMPTY) .,
EX1 = EX2 ('+' EX1 / .EMPTY) .,
```

EX is an abbreviation for expression. The last equation, which defines an expression of order 1, is considered the main equation. The equations are read in this manner. An expression of order 3 is defined as an identifier or an open parenthesis followed by an expression of order 1 followed by a closed parenthesis. An expression of order 2 is defined as an expression of order 3, which may be followed by a star which is followed by an expression of order 2. An expression of order 1 is defined as an expression of order 2, which may be followed by a plus which is followed by an expression of order 1.

Although sequences can be defined recursively, it is more convenient and efficient to have a special operator for this purpose. For example, we can define a sequence of the letter A as follows:

```
SEQA = $ 'A' .,
```

The equations given previously are rewritten using the sequence operator as follows:

```
EX3 = .ID / '(' EX1 ')' .,
EX2 = EX3 $ ('*' EX3) .,
EX1 = EX2 $ ('+' EX2) .,
```

Output

Up to this point we have considered the notation in META II which describes object language syntax. To produce a compiler, output commands are inserted into the syntax equations. Output from a compiler written in META II is always in an assembly language, but not in the assembly language for the 1401. It is for an interpreter, such as the interpreter I call the META II machine, which is used for all compilers, or the interpreters I call the VALGOL I and VALGOL II machines, which obviously are used with their respective source languages. Each machine requires its own assembler, but the main difference between the assemblers is the operation code table. Constant codes and declarations may also be different. These assemblers all have the same format, which is shown below.

```
LABEL     CODE      ADDRESS

1-   -6   8-  -10   12-                          -70
```

An assembly language record contains either a label or an op code of up to 3 characters, but never both. A label begins in column 1 and may extend as far as column 70. If a record contains an op code, then column 1 must be blank. Thus labels may be any length and are not attached to instructions, but occur between instructions.

To produce output beginning in the op code field, we write .OUT and then surround the information to be reproduced with parentheses. A string is used for literal output and an asterisk to output the special symbol just found in the input. This is illustrated as follows:

```
EX3 = .ID .OUT('LD ' *) / '(' EX1 ')' .,
EX2 = EX3 $ ('*' EX3 .OUT('MLT')) .,
EX1 = EX2 $ ('+' EX2 .OUT('ADD')) .,
```

To cause output in the label field we write .LABEL followed by the item to be output. For example, if we want to test for an identifier and output it in the label field we write:

```
.ID .LABEL *
```

The META II compiler can generate labels of the form A01, A02, A03, ... A99, B01, ... To cause such a label to be generated, one uses *1 or *2. The first time *1 is referred to in any syntax equation, a label will be generated and assigned to it. This same label is output whenever *1 is referred to within that execution of the equation. The symbol *2 works in the same way. Thus a maximum of two different labels may be generated for each execution of any equation. Repeated executions, whether recursive or externally initiated, result in a continued sequence of generated labels. Thus all syntax equations contribute to the one sequence. A typical example in which labels are generated for branch commands is now given.

```
IFSTATEMENT = '.IF' EXP '.THEN' .OUT('BFP' *1)
     STATEMENT '.ELSE' .OUT('B ' *2) .LABEL *1
     STATEMENT .LABEL *2 .,
```

The op codes BFP and B are orders of the VALGOL I machine, and stand for "branch false and pop" and "branch" respectively. The equation also contains references to two other equations which are not explicitly given, viz., EXP and STATEMENT.

VALGOL I - A Simple Compiler Written in META II

Now we are ready for an example of a compiler written in META II. VALGOL I is an extremely simple language, based on ALGOL 60, which has been designed to illustrate the META II compiler.

The basic information about VALGOL I is given in figure 1 (the VALGOL I compiler written in META II) and figure 2 (order list of the VALGOL I machine). A sample program is given in figure 3. After each line of the program, the VALGOL I commands which the compiler produces from that line are shown, as well as the absolute interpretive language produced by the assembler. Figure 4 is output from the sample program. Let us study the compiler written in META II (figure 1) in more detail.

The identifier PROGRAM on the first line indicates that this is the main equation, and that control goes there first. The equation for PRIMARY is similar to that of EX3 in our previous example, but here numbers are recognized and reproduced with a "load literal" command. TERM is what was previously EX2; and EXP1 what was previously EX1 except for recognizing minus for subtraction. The equation EXP defines the relational operator "equal", which produces a value of 0 or 1 by making a comparison. Notice that this is handled just like the arithmetic operators but with a lower precedence. The conditional branch commands, "branch true and pop" and "branch false and pop", which are produced by the equations defining UNTILST and CONDITIONALST respectively, will test the top item in the stack and branch accordingly.

The "assignment statement" defined by the equation for ASSIGNST is reversed from the convention in ALGOL 60, i.e., the location into which the computed value is to be stored is on the right. Notice also that the equal sign is used for the assignment statement and that period equal (.=) is used for the relation discussed above. This is because assignment statements are more numerous in typical programs than equal compares, and so the simpler representation is chosen for the more frequently occurring.

The omission of statement labels from the VALGOL I and VALGOL II seems strange to most programmers. This was not done because of any difficulty in their implementation, but because of a dislike for statement labels on the part of the author. I have programmed for several years without using a single label, so I know that they are superfluous from a practical, as well as from a theoretical, standpoint. Nevertheless, it would be too much of a digression to try to justify this point here. The "until statement" has been added to facilitate writing loops without labels.

The "conditional" statement is similar to the one in ALGOL 60, but here the "else" clause is required.

The equation for "input/output", IOST, involves two commands, "edit" and "print". The words EDIT and PRINT do not begin with periods so that they will look like subroutines written in code. "EDIT" copies the given string into the print area, with the first character in the print position which is computed from the given expression. "PRINT" will print the current contents of the print area and then clear it to blanks. Giving a print command without previous edit commands results in writing a blank line.

IDSEQ1 and IDSEQ are given to simplify the syntax equation for DEC (declaration). Notice in the definition of DEC that a branch is given around the data.

From the definition of BLOCK it can be seen that what is considered a compound statement in ALGOL 60 is, in VALGOL I, a special case of a block which has no declaration.

In the definition of statement, the test for an IOST precedes that for an ASSIGNST. This is necessary, because if this were not done the words PRINT and EDIT would be mistaken as identifiers and the compiler would try to translate "input/output" statements as if they were "assignment" statements.

Notice that a PROGRAM is a block and that a standard set of commands is output after each program. The "halt" command causes the machine to stop on reaching the end of the outermost block, which is the program. The operation code SP is generated after the "halt" command. This is a completely 1401-oriented code, which serves to set a word mark at the end of the program. It would not be used if VALGOL I were implemented on a fixed word-length machine.

How the META II Compiler Was Written

Now we come to the most interesting part of this project, and consider how the META II compiler was written in its own language. The interpreter called the META II machine is not a much longer 1401 program than the VALGOL I machine. The syntax equations for META II (figure 5) are fewer in number than those for the VALGOL I machine (figure 1).

The META II compiler, which is an interpretive program for the META II machine, takes the syntax equations given in figure 5 and produces an assembly language version of this same interpretive program. Of course, to get this started, I had to write the first compiler-writing compiler by hand. After the program was running, it could produce the same program as written by hand. Someone always asks if the compiler really produced exactly the program I had written by hand and I have to say that it was "almost" the same program. I followed the syntax equations and tried to write just what the compiler was going to produce. Unfortunately I forgot one of the redundant instructions, so the results were not quite the same. Of course, when the first machine-produced compiler compiled itself the second time, it reproduced itself exactly.

The compiler originally written by hand was for a language called META I. This was used to implement the improved compiler for META II. Sometimes, when I wanted to change the metalanguage, I could not describe the new metalanguage directly in the current metalanguage. Then an intermediate language was created -- one which could be described in the current language and in which the new language could be described. I thought that it might sometimes be necessary to modify the assembly language output, but it seems that it is always possible to avoid this with the intermediate language.

The order list of the META II machine is given in figure 6.

All subroutines in META II programs are recursive. When the program enters a subroutine a stack is pushed down by three cells. One cell is for the exit address and the other two are for labels which may be generated during the execution of the subroutine. There is a switch which may be set or reset by the instructions which refer to the input string, and this is the switch referred to by the conditional branch commands.

The first thing in any META II machine program is the address of the first instruction. During the initialization for the interpreter, this address is placed into the instruction counter.

VALGOL II Written in META II

VALGOL II is an expansion of VALGOL I, and serves as an illustration of a fairly elaborate programming language implemented in the META II system. There are several features in the VALGOL II machine which were not present in the

VALGOL I machine, and which require some explanation. In the VALGOL II machine, addresses as well as numbers are put in the stack. They are marked appropriately so that they can be distinguished at execution time.

The main reason that addresses are allowed in the stack is that, in the case of a subscripted variable, an address is the result of a computation. In an assignment statement each left member is compiled into a sequence of code which leaves an address on top of the stack. This is done for simple variables as well as subscripted variables, because the philosophy of this compiler writing system has been to compile everything in the most general way. A variable, simple or subscripted, is always compiled into a sequence of instructions which leaves an address on top of the stack. The address is not replaced by its contents until the actual value of the variable is needed, as in an arithmetic expression.

A formal parameter of a procedure is stored either as an address or as a value which is computed when the procedure is called. It is up to the load command to go through any number of indirect address in order to place the address of a number onto the stack. An argument of a procedure is always an algebraic expression. In case this expression is a variable, the value of the formal parameter will be an address computed upon entering the procedure; otherwise, the value of the formal parameter will be a number computed upon entering the procedure.

The operation of the load command is now described. It causes the given address to be put on top of the stack. If the content of this top item happens to be another address, then it is replaced by that other address. This continues until the top item on the stack is the address of something which is not an address. This allows for formal parameters to refer to other formal parameters to any depth.

No distinction is made between integer and real numbers. An integer is just a real number whose digits right of the decimal point are zero. Variables initially have a value called "undefined", and any attempt to use this value will be indicated as an error.

An assignment statement consists of any number of left parts followed by a right part. For each left part there is compiled a sequence of commands which puts an address on top of the stack. The right part is compiled into a sequence of instructions which leaves on top of the stack either a number or the address of a number. Following the instruction for the right part there is a sequence of store commands, one for each left part. The first command of this sequence is "save and store", and the rest are "plain" store commands. The "save and store" puts the number which is on top of the stack (or which is referred to by the address on top of the stack) into a register called SAVE. It then stores the contents of SAVE in the address which is held in the next to top position of the stack. Finally it pops the top two items, which it has used, out of the stack. The number, however, remains in SAVE for use by the following store commands. Most assignment statements have only one left part, so "plain" store commands are seldom produced, with the result that the number put in SAVE is seldom used again.

The method for calling a procedure can be explained by reference to illustrations 1 and 2. The arguments which are in the stack are moved to their place at the top of the procedure. If the

```
XXXXXXXX  Function

XXXXXXXX  Arguments
XXXXXXXX
........
XXXXXXXX

b          Word of one blank char-
           acter to mark the end
           of the arguments.

........   Body. Branch commands
........   cause control to go
........   around data stored in
           this area. Ends with
R          a "return" command.
```

Illustration 1

Storage Map for VALGOL II Procedures

```
XXXXXXXX  Arguments in reverse order
XXXXXXXX
........
XXXXXXXX
     XXX  Flag
     XXX  Address of         Exit                XXX
........    procedure                       ........
........                                    ........
Stack before executing      Stack after executing
the call instruction        the call instruction
```

Illustration 2

Map of the Stack Relating to Procedure Calls

number of arguments in the stack does not correspond to the number of arguments in the procedure, an error is indicated. The "flag" in the stack works like this. In the VALGOL II machine there is a flag register. To set a flag in the stack, the contents of this register is put on top of the stack, then the address of the word above the top of the stack is put into the flag register. Initially, and whenever there are no flags in the stack, the flag register contains blanks. At other times it contains the address of the word in the stack which is just above the uppermost flag. Just before a call instruction is executed, the flag register contains the address of the word in the stack which is two above the word containing the address of the procedure to be executed. The call instruction picks up the arguments from the stack, beginning with the one stored just above the flag, and continuing to the top of the stack. Arguments are moved into the appropriate places at the top of the procedure being called. An error message is given if the number of arguments in the stack does not correspond to the number of places in the procedure. Finally the old flag address, which is just below the procedure address in the stack, is put in the flag register. The exit address replaces the address of the procedure in the stack, and all the arguments, as well as the flag, are popped out. There are just two op codes which affect the flag register. The code "load flag" puts a flag into the stack, and the code "call" takes one out.

The library function "WHOLE" truncates a real number. It does not convert a real number to an integer, because no distinction is made between them. It is substituted for the recommended function "ENTIER" primarily because truncation takes fewer machine instructions to implement. Also, truncation seems to be used more frequently. The procedure ENTIER can be defined in VALGOL II as follows:

```
.PROCEDURE ENTIER(X) .,
  .IF 0 .L= X .THEN WHOLE (X) .ELSE
  .IF WHOLE(X) = X .THEN X .ELSE
  WHOLE(X) -1
```

The "for statement" in VALGOL II is not the same as it is in ALGOL. Exactly one list element is required. The "step .. until" portion of the element is mandatory, but the "while" portion may be added to terminate the loop immediately upon some condition. The iteration continues so long as the value of the variable is less than or equal to the maximum, irrespective of the sign of the increment. Illustration 3 is an example of a typical "for statement". A flow chart of this statement is given in illustration 4.

```
.FOR I = 0 .STEP 1 .UNTIL N .DO
      <statement>
      SET               Set switch to indicate first
A91                     time through.
      LD    I
      FLP   ]
      BFP   A92]        Test for first time through.
      LDL   0  ]
      SST      ]        Initialize variable.
      B     A93]
A92
      LDL   1  ]        Increment variable.
      ADS      ]
A93
      RSR      ]        Compare variable to maximum.
      LD    N  ]
      LEQ      ]
      BFP   A94]
      <statement>
      RST               Reset switch to indicate not
                        first time through.
      B     A91
A94
```

Illustration 3
Compilation of a typical "for statement" in VALGOL II

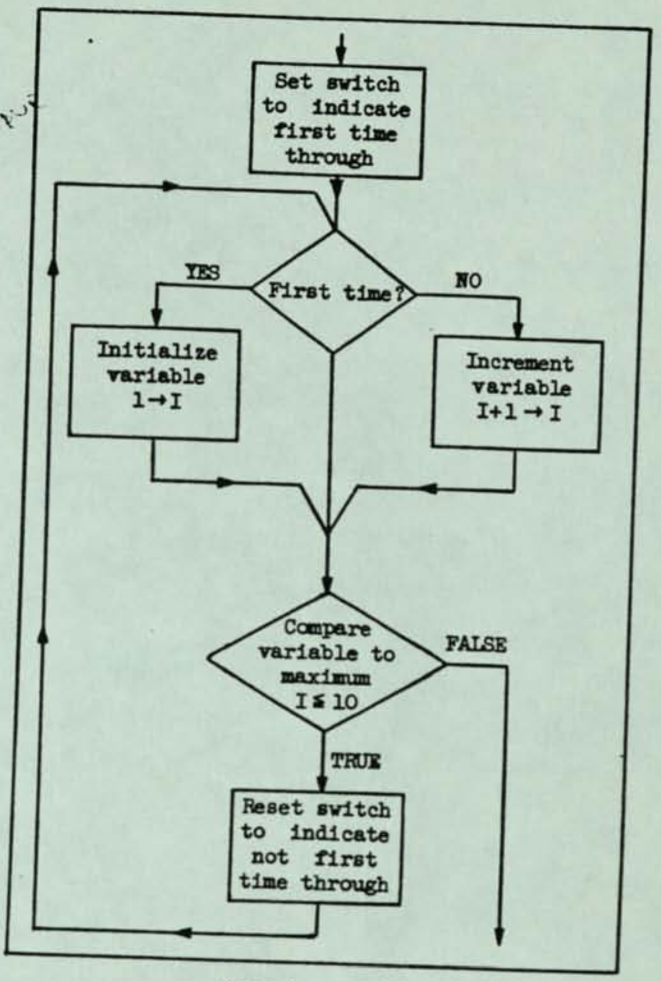


Illustration 4

Flow chart of the "for statement" given in figure 12

Figure 7 is a listing of the VALGOL II compiler written in META II. Figure 8 gives the order list of the VALGOL II machine. A sample program to take a determinant is given in figure 9.

Backup vs. No Backup

Suppose that, upon entry to a recursive subroutine, which represents some syntax equation, the position of the input and output are saved. When some non-first term of a component is not found, the compiler does not have to stop with an indication of a syntax error. It can back-up the input and output and return false. The advantages of backup are as follows:

1. It is possible to describe languages, using backup, which cannot be described without backup.

2. Even for a language which can be described without backup, the syntax equations can often be simplified when backup is allowed.

The advantages claimed for non-backup are as follows:

1. Syntax analysis is faster.

2. It is possible to tell whether syntax equations will work just by examining them, without following through numerous examples.

The fact that rather sophisticated languages such as ALGOL and COBOL can be implemented without backup is pointed out by various people, including Conway,[5] and they are aware of the speed advantages of so doing. I have seen no mention of the second advantage of no-backup, so I will explain this in more detail.

Basically one writes alternations in which each term begins with a different symbol. Then it is not possible for the compiler to go down the wrong path. This is made more complicated because of the use of ".EMPTY". An optional item can never be followed by something that begins with the same symbol it begins with.

The method described above is not the only way in which backup can be handled. Variations are worth considering, as a way may be found to have the advantages of both backup and no-backup.

Further Development of META Languages

As mentioned earlier, META II is not presented as a standard language, but as a point of departure from which a user may develop his own META language. The term "META Language," with "META" in capital letters, is used to denote any compiler-writing language so developed.

The language which Schmidt[1] implemented on the PDP-1 was based on META I. He has now implemented an improved version of this language for a Beckman machine.

Rutman[9] has implemented LOGIK, a compiler for bit-time simulation, on the 7090. He uses a META language to compile Boolean expressions into efficient machine code. Schneider and Johnson[10] have implemented META 3 on the IBM 7094, with the goal of producing an ALGOL compiler which generates efficient machine code. They are planning a META language which will be suitable for any block structured language. To this compiler-writing language they give the name META 4 (pronounced metaphor).

References

1. Schmidt, L., "Implementation of a Symbol Manipulator for Heuristic Translation," 1963 ACM Natl. Conf., Denver, Colo.

2. Metcalfe, Howard, "A Parameterized Compiler Based on Mechanical Linguistics," 1963 ACM Natl. Conf., Denver, Colo.

3. Schorre, Val, "A Syntax - Directed SMALGOL for the 1401," 1963 ACM Natl. Conf., Denver, Colo.

4. Glennie, A., "On the Syntax Machine and the Construction of a Universal Compiler," Tech. Report No. 2, Contract NR 049-141, Carnegie Inst. of Tech., July, 1960.

5. Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler," Comm. ACM, July 1963.

6. Irons, E. T., The Structure and Use of the Syntax - Directed Compiler," Annual Review in Automatic Programming, The Macmillan Co., New York.

7. Bastian, Lewis, "A Phrase-Structure Language Translator," AFCRL-Rept-62-549, Aug. 1962.

8. Chomsky, Noam, "Syntax Structures," Mouton and Co., Publishers, The Hague, Netherlands.

9. Rutman, Roger, "LOGIK, A Syntax Directed Compiler for Computer Bit-Time Simulation," Master Thesis, UCLA, August 1964.

10. Schneider, F. W., and G. D. Johnson, "A Syntax-Directed Compiler-Writing Compiler to Generate Efficient Code," 1964 ACM Natl. Conf., Philadelphia.

THE VALGOL I COMPILER WRITTEN IN META II LANGUAGE
FIGURE 1

```
.SYNTAX PROGRAM

PRIMARY = .ID .OUT('LD ' *) /
     .NUMBER .OUT('LDL' *) /
     '(' EXP ')' .,

TERM = PRIMARY $('*' PRIMARY .OUT('MLT') ) .,

EXP1 = TERM $('+' TERM .OUT('ADD') /
     '-' TERM .OUT('SUB') ) .,

EXP = EXP1 ( '.=' EXP1 .OUT('EQU') / .EMPTY) .,

ASSIGNST = EXP '=' .ID .OUT('ST ' *) .,

UNTILST = '.UNTIL' .LABEL *1 EXP '.DO' .OUT('BTP' *2)
     ST .OUT('B  ' *1) .LABEL *2 .,

CONDITIONALST = '.IF' EXP '.THEN' .OUT('BFP' *1)
     ST '.ELSE' .OUT('B  ' *2) .LABEL *1
     ST .LABEL *2 .,

IOST = 'EDIT' '(' EXP ',' .STRING
     .OUT('EDT' *) ')' /
     'PRINT' .OUT('PNT') .,

IDSEQ1 = .ID .LABEL * .OUT('BLK 1') .,

IDSEQ = IDSEQ1 $(',' IDSEQ1) .,

DEC = '.REAL' .OUT('B  ' *1) IDSEQ .LABEL *1 .,

BLOCK = '.BEGIN' (DEC '.,' / .EMPTY)
     ST $('.,' ST) '.END' .,

ST = IOST / ASSIGNST / UNTILST /
     CONDITIONALST / BLOCK .,

PROGRAM = BLOCK .OUT('HLT')
     .OUT('SP  1') .OUT('END') .,

.END
```

ORDER LIST OF THE VALGOL I MACHINE
FIGURE 2

MACHINE CODES

LD AAA	LOAD	PUT THE CONTENTS OF THE ADDRESS AAA ON TOP OF THE STACK.
LDL NUMBER	LOAD LITERAL	PUT THE GIVEN NUMBER ON TOP OF THE STACK.
ST AAA	STORE	STORE THE NUMBER WHICH IS ON TOP OF THE STACK INTO THE ADDRESS AAA AND POP UP THE STACK.
ADD	ADD	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR SUM.
SUB	SUBTRACT	SUBTRACT THE NUMBER WHICH IS ON TOP OF THE STACK FROM THE NUMBER WHICH IS NEXT TO THE TOP, THEN REPLACE THEM BY THIS DIFFERENCE.
MLT	MULTIPLY	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR PRODUCT.
EQU	EQUAL	COMPARE THE TWO NUMBERS ON TOP OF THE STACK. REPLACE THEM BY THE INTEGER 1, IF THEY ARE EQUAL, OR BY THE INTEGER 0, IF THEY ARE UNEQUAL.
B AAA	BRANCH	BRANCH TO THE ADDRESS AAA.
BFP AAA	BRANCH FALSE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
BTP AAA	BRANCH TRUE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
EDT STRING	EDIT	ROUND THE NUMBER WHICH IS ON TOP OF THE STACK TO THE NEAREST INTEGER N. MOVE THE GIVEN STRING INTO THE PRINT AREA SO THAT ITS FIRST CHARACTER FALLS ON PRINT POSITION N. IN CASE THIS WOULD CAUSE CHARACTERS TO FALL OUTSIDE THE PRINT AREA, NO MOVEMENT TAKES PLACE.
PNT	PRINT	PRINT A LINE, THEN SPACE AND CLEAR THE PRINT AREA.
HLT	HALT	HALT.

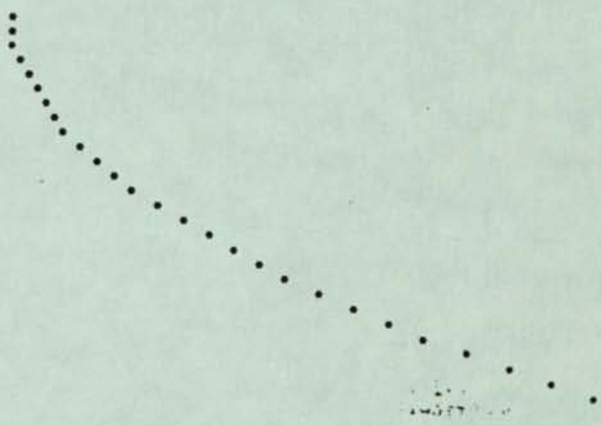
CONSTANT AND CONTROL CODES

SP N	SPACE	N = 1--9. CONSTANT CODE PRODUCING N BLANK SPACES.
BLK NNN	BLOCK	PRODUCES A BLOCK OF NNN EIGHT CHARACTER WORDS.
END	END	DENOTES THE END OF THE PROGRAM.

A PROGRAM AS COMPILED FOR THE VALGOL I MACHINE
FIGURE 3

```
.BEGIN
.REAL X ., 0 = X .,
             B   A01                               0000 G 0012
     X                                             0004
             BLK 001                               0004
     A01                                           0012
             LDL 0                                 0012 A
             ST  X                                 0021 B 0004
.UNTIL X .= 3 .DO .BEGIN
     A02                                           0025
             LD  X                                 0025 0 0004
             LDL 3                                 0029 A
             EQU                                   0038 F
             BTP A03                               0039 K 0097
   EDIT( X*X * 10 + 1, '*') ., PRINT ., X + 0.1 = X
             LD  X                                 0043 0 0004
             LD  X                                 0047 0 0004
             MLT                                   0051 E
             LDL 10                                0052 A
             MLT                                   0061 E
             LDL 1                                 0062 A
             ADD                                   0071 C
             EDT 01'*'                             0072 I
             PNT                                   0074 0
             LD  X                                 0075 0 0004
             LDL 0.1                               0079 A
             ADD                                   0088 C
             ST  X                                 0089 B 0004
   .END
             B   A02                               0093 G 0025
      A03                                          0097
.END
             HLT                                   0097 J
             SP  1                                 0098
             END                                   0099
```

OUTPUT FROM THE VALGOL I PROGRAM GIVEN IN FIGURE 3
FIGURE 4



THE META II COMPILER WRITTEN IN ITS OWN LANGUAGE
FIGURE 5

```
.SYNTAX PROGRAM

OUT1 = '*1' .OUT('GN1') / '*2' .OUT('GN2') /
'*' .OUT('CI') / .STRING .OUT('CL ' *).,

OUTPUT = ('.OUT' '('
$ OUT1 ')' / '.LABEL' .OUT('LB') OUT1) .OUT('OUT') .,

EX3 = .ID .OUT ('CLL' *) / .STRING
.OUT('TST' *) / '.ID' .OUT('ID') /
'.NUMBER' .OUT('NUM') /
'.STRING' .OUT('SR') / '(' EX1 ')' /
'.EMPTY' .OUT('SET') /
'$' .LABEL *1 EX3
.OUT ('BT ' *1) .OUT('SET').,

EX2 = (EX3 .OUT('BF ' *1) / OUTPUT)
$(EX3 .OUT('BE') / OUTPUT)
.LABEL *1 .,

EX1 = EX2 $('/' .OUT('BT ' *1) EX2 )
.LABEL *1 .,

ST = .ID .LABEL * '=' EX1
'.,' .OUT('R').,

PROGRAM = '.SYNTAX' .ID .OUT('ADR' *)
$ ST '.END' .OUT('END').,

.END
```

ORDER LIST OF THE META II MACHINE
FIGURE 6

MACHINE CODES

TST STRING	TEST	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, COMPARE IT TO THE STRING GIVEN AS ARGUMENT. IF THE COMPARISON IS MET, DELETE THE MATCHED PORTION FROM THE INPUT AND SET SWITCH. IF NOT MET, RESET SWITCH.
ID	IDENTIFIER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH AN IDENTIFIER, IE., A LETTER FOLLOWED BY A SEQUENCE OF LETTERS AND/OR DIGITS. IF SO, DELETE THE IDENTIFIER AND SET SWITCH. IF NOT, RESET SWITCH.
NUM	NUMBER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A NUMBER. A NUMBER IS A STRING OF DIGITS WHICH MAY CONTAIN IMBEDED PERIODS, BUT MAY NOT BEGIN OR END WITH A PERIOD. MOREOVER, NO TWO PERIODS MAY BE NEXT TO ONE ANOTHER. IF A NUMBER IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
SR	STRING	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A STRING, IE., A SINGLE QUOTE FOLLOWED BY A SEQUENCE OF ANY CHARACTERS OTHER THAN SINGLE QUOTE FOLLOWED BY ANOTHER SINGLE QUOTE. IF A STRING IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
CLL AAA	CALL	ENTER THE SUBROUTINE BEGINNING IN LOCATION AAA. IF THE TOP TWO TERMS OF THE STACK ARE BLANK, PUSH THE STACK DOWN BY ONE CELL. OTHERWISE, PUSH IT DOWN BY THREE CELLS. SET A FLAG IN THE STACK TO INDICATE WHETHER IT HAS BEEN PUSHED BY ONE OR THREE CELLS. THIS FLAG AND THE EXIT ADDRESS GO INTO THE THIRD CELL. CLEAR THE TOP TWO CELLS TO BLANKS TO INDICATE THAT THEY CAN ACCEPT ADDRESSES WHICH MAY BE GENERATED WITHIN THE SUBROUTINE.

Figure 6.1

R	RETURN	RETURN TO THE EXIT ADDRESS, POPPING UP THE STACK BY ONE OR THREE CELLS ACCORDING TO THE FLAG. IF THE STACK IS POPPED BY ONLY ONE CELL, THEN CLEAR THE TOP TWO CELLS TO BLANKS, BECAUSE THEY WERE BLANK WHEN THE SUBROUTINE WAS ENTERED.
SET	SET	SET BRANCH SWITCH ON.
B AAA	BRANCH	BRANCH UNCONDITIONALLY TO LOCATION AAA.
BT AAA	BRANCH IF TRUE	BRANCH TO LOCATION AAA IF SWITCH IS ON. OTHERWISE, CONTINUE IN SEQUENCE.
BF AAA	BRANCH IF FALSE	BRANCH TO LOCATION AAA IF SWITCH IS OFF. OTHERWISE, CONTINUE IN SEQUENCE.
BE	BRANCH TO ERROR IF FALSE	HALT IF SWITCH IS OFF. OTHERWISE, CONTINUE IN SEQUENCE.
CL STRING	COPY LITERAL	OUTPUT THE VARIABLE LENGTH STRING GIVEN AS THE ARGUMENT. A BLANK CHARACTER WILL BE INSERTED IN THE OUTPUT FOLLOWING THE STRING.
CI	COPY INPUT	OUTPUT THE LAST SEQUENCE OF CHARACTERS DELETED FROM THE INPUT STRING. THIS COMMAND MAY NOT FUNCTION PROPERLY IF THE LAST COMMAND WHICH COULD CAUSE DELETION FAILED TO DO SO.
GN1	GENERATE 1	THIS CONCERNS THE CURRENT LABEL 1 CELL, IE., THE NEXT TO TOP CELL IN THE STACK, WHICH IS EITHER CLEAR OR CONTAINS A GENERATED LABEL. IF CLEAR, GENERATE A LABEL AND PUT IT INTO THAT CELL. WHETHER THE LABEL HAS JUST BEEN PUT INTO THE CELL OR WAS ALREADY THERE, OUTPUT IT. FINALLY, INSERT A BLANK CHARACTER IN THE OUTPUT FOLLOWING THE LABEL.
GN2	GENERATE 2	SAME AS GN1, EXCEPT THAT IT CONCERNS THE CURRENT LABEL 2 CELL, IE., THE TOP CELL IN THE STACK.
LB	LABEL	SET THE OUTPUT COUNTER TO CARD COLUMN 1.
OUT	OUTPUT	PUNCH CARD AND RESET OUTPUT COUNTEI TO CARD COLUMN 8.

Figure 6.2

CONSTANT AND CONTROL CODES

ADR IDENT	ADDRESS	PRODUCES THE ADDRESS WHICH IS ASSIGNED TO THE GIVEN IDENTIFIER , A CONSTANT.
END	END	DENOTES THE END OF THE PROGRAM.

Figure 6.3

VALGOL II COMPILER WRITTEN IN META II
FIGURE 7

```
.SYNTAX PROGRAM

ARRAYPART = '(.' EXP '.)' .OUT('AIA') ..

CALLPART = '(' .OUT('LDF') (EXP $(',' EXP) /
     .EMPTY) ')' .OUT('CLL') ..

VARIABLE = .ID .OUT('LD ' *) (ARRAYPART / .EMPTY) ..

PRIMARY = 'WHOLE' '(' EXP ')' .OUT('WHL') /
     .ID .OUT('LD ' *) (ARRAYPART / CALLPART / .EMPTY) /
     '.TRUE' .OUT('SET') / '.FALSE' .OUT('RST') /
     '0 ' .OUT('RST') / '1 ' .OUT('SET') /
     .NUMBER .OUT('LDL' *) /
     '(' EXP ')' ..

TERM = PRIMARY $ ('*' PRIMARY .OUT('MLT') /
     '/' PRIMARY .OUT('DIV') /
     './.' PRIMARY .OUT('DIV') .OUT('WHL') ) ..

EXP2 = '-' TERM .OUT('NEG') /
     '+' TERM / TERM ..

EXP1 = EXP2 $('+' TERM .OUT('ADD') /
     '-' TERM .OUT('SUB')) ..

RELATION = EXP1 (
     '.L=' EXP1 .OUT('LEQ') /
     '.L' EXP1 .OUT('LES') /
     '.=' EXP1 .OUT('EQU') /
     '.-=' EXP1 .OUT('EQU') .OUT('NOT') /
     '.G=' EXP1 .OUT('LES') .OUT('NOT') /
     '.G' EXP1 .OUT('LEQ') .OUT('NOT') /
     .EMPTY) ..

BPRIMARY = '.-' RELATION .OUT('NOT') /
     RELATION ..

BTERM = BPRIMARY $ ('.A' .OUT('BF ' *1)
     .OUT('POP') BPRIMARY)
     .LABEL *1 ..

BEXP1 = BTERM $( '.V' .OUT('BT ' *1)
     .OUT('POP') BTERM)
     .LABEL *1 ..

IMPLICATION1 = '.IMP' .OUT('NOT')
     .OUT('BT ' *1) .OUT('POP')
     BEXP1 .LABEL *1 ..

/ IMPLICATION = BEXP1 $ IMPLICATION1 ..
```

Figure 7.1

```
EQUIV = IMPLICATION $('.EQ' .OUT('EQU') ) ..

EXP = '.IF' EXP '.THEN' .OUT('BFP' *1)
     EXP .OUT('B  ' *2) .LABEL *1
     '.ELSE' EXP .LABEL *2 /
     EQUIV ..

ASSIGNPART = '=' EXP ( ASSIGNPART .OUT('ST') /
     .EMPTY .OUT('SST') ) ..

ASSIGNCALLST = .ID .OUT('LD ' *) (ARRAYPART ASSIGNPART /
     ASSIGNPART / (CALLPART / .EMPTY
     .OUT('LDF') .OUT('CLL') )
     .OUT('POP') ) ..

UNTILST = '.UNTIL' .LABEL *1 EXP
     '.DO' .OUT('BTP' *2) ST
     .OUT('B  ' *1) .LABEL *2 ..

WHILECLAUSE = '.WHILE' .OUT('BF ' *1)
     .OUT('POP') EXP .LABEL *1 / .EMPTY ..

FORCLAUSE = VARIABLE '=' .OUT('FLP')
     .OUT('BFP' *1) EXP '.STEP'
     .OUT('SST') .OUT('B  ' *2)
     .LABEL *1 EXP '.UNTIL' .OUT('ADS')
     .LABEL *2 .OUT('RSR') EXP
     .OUT('LEQ') WHILECLAUSE '.DO' ..

FORST = '.FOR' .OUT('SET') .LABEL *1
     FORCLAUSE .OUT('BFP' *2) ST
     .OUT('RST') .OUT('B  ' *1)
     .LABEL *2 ..

IOCALL = 'READ' '(' VARIABLE ',' EXP ')' .OUT('RED') /
     'WRITE' '(' VARIABLE ',' EXP ')' .OUT('WRT') /
     'EDIT' '(' EXP ',' .STRING
     .OUT('EDT' *) ')' /
     'PRINT' .OUT('PNT') /
     'EJECT' .OUT('EJT') ..

IDSEQ1 = .ID .LABEL* .OUT('BLK 1') ..

IDSEQ = IDSEQ1 $(',' IDSEQ1) ..

TYPEDEC = '.REAL' IDSEQ ..

ARRAY1 = .ID .LABEL * '(.' '0' '..' .NUMBER
     .OUT('BLK 1') .OUT('BLK' *) '.)' ..

ARRAYDEC = '.ARRAY' ARRAY1 $( ',' ARRAY1) ..

PROCEDURE = '.PROCEDURE' .ID .LABEL *
     .LABEL *1 .OUT('BLK 1') '('
     (IDSEQ / .EMPTY) ')' .OUT('SP  1') '.,'
     [illegible] *1) ..
```

Figure 7.2

```
DEC = TYPEDEC / ARRAYDEC / PROCEDURE ..

BLOCK = '.BEGIN' .OUT('B  ' *1) $(DEC '.,')
     .LABEL *1 ST $('.,' ST) '.END'
     (.ID / .EMPTY) ..

UNCONDITIONALST = IOCALL / ASSIGNCALLST /
     BLOCK ..

CONDST = '.IF' EXP '.THEN' .OUT('BFP' *1)
     (UNCONDITIONALST ('.ELSE' .OUT('B  ' *2)
     .LABEL *1 ST .LABEL *2 / .EMPTY
     .LABEL *1) / (FORST / UNTILST)
     .LABEL *1) ..

ST = CONDST / UNCONDITIONALST / FORST /
     UNTILST / .EMPTY ..

PROGRAM = BLOCK
     .OUT('HLT') .OUT('SP  1') .OUT('END') ..

.END
```

Figure 7.3

ORDER LIST OF THE VALGOL II MACHINE
FIGURE 8

MACHINE CODES

LD AAA	LOAD	PUT THE ADDRESS AAA ON TOP OF THE STACK.
LDL NUMBER	LOAD LITERAL	PUT THE GIVEN NUMBER ON TOP OF THE STACK.
SET	SET	PUT THE INTEGER 1 ON TOP OF THE STACK.
RST	RESET	PUT THE INTEGER 0 ON TOP OF THE STACK.
ST	STORE	STORE THE CONTENTS OF THE REGISTER, STACK1, IN THE ADDRESS WHICH IS ON TOP OF THE STACK, THEN POP UP THE STACK.
ADS	ADD TO STORAGE NOTE 1	ADD THE NUMBER ON TOP OF THE STACK TO THE NUMBER WHOSE ADDRESS IS NEXT TO THE TOP, AND PLACE THE SUM IN THE REGISTER, STACK1. THEN STORE THE CONTENTS OF THAT REGISTER IN THAT ADDRESS, AND POP THE TOP TWO ITEMS OUT OF THE STACK.
SST	SAVE AND STORE NOTE 1	PUT THE NUMBER ON TOP OF THE STACK INTO THE REGISTER, STACK1. THEN STORE THE CONTENTS OF THAT REGISTER IN THE ADDRESS WHICH IS THE NEXT TO TOP TERM OF THE STACK. THE TOP TWO ITEMS ARE POPPED OUT OF THE STACK.
RSR	RESTORE	PUT THE CONTENTS OF THE REGISTER, STACK1, ON TOP OF THE STACK.
ADD	ADD NOTE 2	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR SUM.
SUB	SUBTRACT NOTE 2	SUBTRACT THE NUMBER WHICH IS ON TOP OF THE STACK FROM THE NUMBER WHICH IS NEXT TO THE TOP, THEN REPLACE THEM BY THIS DIFFERENCE.
MLT	MULTIPLY NOTE 2	REPLACE THE TWO NUMBERS WHICH ARE ON TOP OF THE STACK WITH THEIR PRODUCT.
DIV	DIVIDE NOTE 2	DIVIDE THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK BY THE NUMBER WHICH IS ON TOP OF THE STACK, THEN REPLACE THEM BY THIS QUOTIENT.

Figure 8.1

NEG	NEGATE NOTE 1	CHANGE THE SIGN OF THE NUMBER ON TOP OF THE STACK.
WHL	WHOLE	TRUNCATE THE NUMBER WHICH IS ON TOP OF THE STACK.
NOT	NOT	IF THE TOP TERM IN THE STACK IS THE INTEGER 0, THEN REPLACE IT WITH THE INTEGER 1, OTHERWISE, REPLACE IT WITH THE INTEGER 0.
LEQ	LESS THAN OR EQUAL NOTE 2	IF THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK IS LESS THAN OR EQUAL TO THE NUMBER ON TOP OF THE STACK, THEN REPLACE THEM WITH THE INTEGER 1. OTHERWISE, REPLACE THEM WITH THE INTEGER 0.
LES	LESS THAN NOTE 2	IF THE NUMBER WHICH IS NEXT TO THE TOP OF THE STACK IS LESS THAN THE NUMBER ON TOP OF THE STACK, THEN REPLACE THEM WITH THE INTEGER 1. OTHERWISE, REPLACE THEM WITH THE INTEGER 0.
EQU	EQUAL NOTE 2	COMPARE THE TWO NUMBERS ON TOP OF THE STACK. REPLACE THEM BY THE INTEGER 1, IF THEY ARE EQUAL, OR BY THE INTEGER 0, IF THEY ARE UNEQUAL.
B AAA	BRANCH	BRANCH TO THE ADDRESS AAA.
BT AAA	BRANCH TRUE	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. DO NOT POP UP THE STACK.
BF AAA	BRANCH FALSE	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. DO NOT POP UP THE STACK.
BTP AAA	BRANCH TRUE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS NOT THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.
BFP AAA	BRANCH FALSE AND POP	BRANCH TO THE ADDRESS AAA IF THE TOP TERM IN THE STACK IS THE INTEGER 0. OTHERWISE, CONTINUE IN SEQUENCE. IN EITHER CASE, POP UP THE STACK.

Figure 8.2

CLL	CALL	ENTER A PROCEDURE AT THE ADDRESS WHICH IS BELOW THE FLAG.
LDF	LOAD FLAG	PUT THE ADDRESS WHICH IS IN THE FLAG REGISTER ON TOP OF THE STACK, AND PUT THE ADDRESS OF THE TOP OF THE STACK INTO THE FLAG REGISTER.
R AAA	RETURN	RETURN FROM PROCEDURE.
AIA	ARRAY INCREMENT ADDRESS	INCREMENT THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK BY THE INTEGER WHICH IS ON TOP OF THE STACK, AND REPLACE THESE BY THE RESULTING ADDRESS.
FLP	FLIP	INTERCHANGE THE TOP TWO TERMS OF THE STACK.
POP	POP	POP UP THE STACK.
EDT STRING	EDIT NOTE 1	ROUND THE NUMBER WHICH IS ON TOP OF THE STACK TO THE NEAREST INTEGER N. MOVE THE GIVEN STRING INTO THE PRINT AREA SO THAT ITS FIRST CHARACTER FALLS ON PRINT POSITION N. IN CASE THIS WOULD CAUSE CHARACTERS TO FALL OUTSIDE THE PRINT AREA, NO MOVEMENT TAKES PLACE.
PNT	PRINT	PRINT A LINE, THEN SPACE AND CLEAR THE PRINT AREA.
EJT	EJECT	POSITION THE PAPER IN THE PRINTER TO THE TOP LINE OF THE NEXT PAGE.
RED	READ	READ THE FIRST N NUMBERS FROM A CARD AND STORE THEM BEGINNING IN THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK. THE INTEGER N IS THE TOP TERM OF THE STACK. POP OUT BOTH THE ADDRESS AND THE INTEGER. CARDS ARE PUNCHED WITH UP TO 10 EIGHT DIGIT NUMBERS. DECIMAL POINT IS ASSUMED TO BE IN THE MIDDLE. AN 11-PUNCH OVER THE RIGHTMOST DIGIT INDICATES A NEGATIVE NUMBER.

Figure 8.3

WRT	WRITE	PRINT A LINE OF N NUMBERS BEGINNING IN THE ADDRESS WHICH IS NEXT TO THE TOP OF THE STACK. THE INTEGER N IS THE TOP TERM OF THE STACK. POP OUT BOTH THE ADDRESS AND THE INTEGER. TWELVE CHARACTER POSITIONS ARE ALLOWED FOR EACH NUMBER. THERE ARE FOUR DIGITS BEFORE AND FOUR DIGITS AFTER THE DECIMAL. LEADING ZEROES IN FRONT OF THE DECIMAL ARE CHANGED TO BLANKS. IF THE NUMBER IS NEGATIVE, A MINUS SIGN IS PRINTED IN THE POSITION BEFORE THE FIRST NON-BLANK CHARACTER.
HLT	HALT	HALT.

CONSTANT AND CONTROL CODES

SP N	SPACE	N = 1--9. CONSTANT CODE PRODUCING N BLANK SPACES.
BLK NNN	BLOCK	PRODUCES A BLOCK OF NNN EIGHT CHARACTER WORDS.
END	END	DENOTES THE END OF THE PROGRAM.

NOTE 1. IF THE TOP ITEM IN THE STACK IS AN ADDRESS, IT IS REPLACED BY ITS CONTENTS BEFORE BEGINNING THIS OPERATION.

NOTE 2. SAME AS NOTE 1, BUT APPLIES TO THE TOP TWO ITEMS.

Figure 8.4

EXAMPLE PROGRAM IN VALGOL II
FIGURE 9

```
.BEGIN
.PROCEDURE DETERMINANT(A, N) .,
.BEGIN
.PROCEDURE DUMP() .,
.BEGIN
.REAL D .,
.FOR D = 0 .STEP 1 .UNTIL N-1 .DO
     WRITE(MATRIX(. N*D .), N) .,
PRINT
.END DUMP .,
.PROCEDURE ABS(X) .,
     ABS = .IF 0 .L= X .THEN X .ELSE -X .,
.REAL PRODUCT, FACTOR, TEMP, R, I, J .,
PRODUCT = 1 .,
.FOR R = 0 .STEP 1 .UNTIL N-2
.WHILE PRODUCT .-= 0 .DO .BEGIN
     I = R .,
     .FOR J = R+1 .STEP 1 .UNTIL N-1 .DO
          .IF ABS( A(. N*I + R .) ) .L
          ABS( A(. N*J + R .) ) .THEN
               I = J .,
     .IF A(. N*I + R .) .= 0 .THEN
          PRODUCT = 0
     .ELSE
          .IF I .-= R .THEN .BEGIN
               PRODUCT = -PRODUCT .,
               .FOR J = R .STEP 1 .UNTIL N-1 .DO
               .BEGIN
                    TEMP = A(. N*R + J .) .,
                    A(. N*R + J .) = A(. N* I + J .) .,
                    A(. N*I + J .) = TEMP .END .END .,
          TEMP = A(. N*R + R .) .,
          .FOR I = R+1 .STEP 1 .UNTIL N-1 .DO
          .BEGIN
               FACTOR = A(. N*I + R .) / TEMP .,
               .FOR J = R .STEP 1 .UNTIL N-1 .DO
                    A(. N*I + J .) = A(. N*I + J .)
                    -FACTOR * A(. N*R + J .) .,
          DUMP
          .END .END .,
.FOR I = 0 .STEP 1 .UNTIL N-1 .DO
     PRODUCT = PRODUCT * A(. N*I + I .) .,
DETERMINANT = PRODUCT
.END DETERMINANT .,
.REAL M, W, T .. .ARRAY MATRIX (. 0 .. 24 .) .,
.UNTIL .FALSE .DO .BEGIN
     EDIT(1, 'FIND DETERMINANT OF' ) ., PRINT., PRINT.,
     READ(M, 1) .,
     .FOR W = 0 .STEP 1 .UNTIL M-1 .DO .BEGIN
          READ(MATRIX (. M*W .), M) .,
          WRITE(MATRIX (. M*W .), M) .END .,
     PRINT .,  T = DETERMINANT (MATRIX, M) .,
     WRITE(T, 1) ., PRINT., PRINT .END
.END PROGRAM
```

Association for Computing Machinery

Proceedings

OF THE 19TH NATIONAL CONFERENCE
PHILADELPHIA, PENNSYLVANIA
AUGUST 25-27, 1964

ACM PUBLICATION P-64

AVAILABLE FROM
ASSOCIATION FOR COMPUTING MACHINERY
211 EAST 43RD STREET
NEW YORK, N.Y. 10017

PRICE $5.00