META PI—An on-line interactive compiler-compiler

by JOHN T. O'NEIL, Jr.

RCA Laboratories
Princeton, New Jersey

It is difficult to specifically date the origin of the research efforts within the programming discipline that are directed at describing and implementing a language which would produce compilers.

The motivation for these efforts stems from meta languages such as Backus Normal Form (BNF)[1] which attempt to describe in a mathematical notation the syntax (structure) of a programming language. The thinking is that if a given language (FORTRAN, ALGOL, etc.) could be described in rather precise form, then it should be possible to construct a translator that would accept statements, say, in BNF and output the appropriate compiler. This processor is shown schematically in Figure 1.

The actual construction of the compiler-compiler has proved to be an elusive goal; the efficient implementation of the theoretically possible turned out to be far more difficult than originally anticipated.

In early 1966 work began at the RCA Laboratories, Princeton, on what has since evolved into RCA BTSS II (Basic Time Sharing System, Version II). During the design discussions for this system it was decided that the interactive language would be based on FORTRAN IV. It was further decided to implement the language, so far as possible, using a compiler-compiler. The final compiler was named FORTRAN PI and its compiler-compiler parent, META PI. It is the opinion of the author based on implementation experience and user acceptance that the viability of the compiler-compiler has been amply demonstrated by the research effort which produced META PI and FORTRAN PI.

Before discussing META PI it will be necessary to discuss BTSS II since it is the operational environment within which META PI functions.

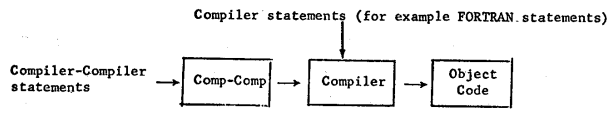

FIGURE 1

RCA BTSS II provides the on-line user the ability to create, modify, execute and correct programs on an interactive basis. The user accesses the system services through three main software components:

A Command Language.
A Text Editor.
The FORTRAN PI compiler and META PI compiler-compiler.

FORTRAN PI and META PI were designed as far as possible to be independent of a given control system and I/O package. Both FORTRAN PI and META PI interface with the system via an interactive executive.

RCA BTSS II is implemented on an RCA SPECTRA 70/45 computer system with 131K of memory. The SPECTRA 70/45 is a third generation computer system with an instruction set which is compatible with System 360. It does not have hardware features (paging, read memory protect, etc.) specifically designed for time sharing. (The RCA SPECTRA 70/46 does have these features, and a version of the PI compiler is operating on it.)

FORTRAN PI was the first language implemented with META PI. A discussion of its design and the structure of the object code produced by it will be helpful in providing the reader insight into the design and function of META PI.

The reader is cautioned to keep in mind the various possible levels of translator activity, that is, the initial creation of FORTRAN PI via META PI, the on-line creation of the user's program via FORTRAN PI, and the on-line creation of the user's compiler (or compiler-compiler).

During the preliminary design phase for BTSS the fundamental decision was made to use a FORTRAN like language as the problem solving language of the system. Three considerations provided the framework for all subsequent design decisions.

First, to gain insight into the viability of compiler-compiler approaches, the implementation of FORTRAN PI would proceed only after the structure of the META PI compiler- compiler was described in detail. As much as possible of the FORTRAN PI compiler would be implemented via META PI.

Second, trade offs would be made in the total META PI approach if, in implementing FORTRAN PI, efficiency of the production compiler would be seriously impaired by this approach.

Third, FORTRAN IV standards would be adhered to wherever possible, but since the language was to be utilized in a time sharing environment, departures from FORTRAN IV standards would be effected whenever the convenience of the terminal user would suffer otherwise.

In retrospect, considering the rather ambitious design constraints, FORTRAN PI was able to meet the bulk of its design objectives. Over 80% of the object code of FORTRAN PI is generated from META PI. The compiler itself is remarkably similar to FORTRAN IV when one considers the conflicts of user utility that arise when one attempts to reconcile a language designed for the batch user with the requirements of interactive time sharing.

For example, some of the present FORTRAN PI alterations to FORTRAN IV are:

1. Free field input format to both compiler and I/O Formatter.
2. Format statements are optional.
3. Recursive functions and subroutines.
4. Arbitrary subscripts.
5. Negative increments in DO loops, etc.
6. Symbolic variable tracing, flow tracing, and other debugging aids.

Further alterations are, of course, easily implemented via META PI.

The FORTRAN PI compiler has the following characteristics:

1. Statements are accepted and compiled a line at a time on an interactive basis.
2. The object code generated is read only and is capable of immediate execution. (The FORTRAN PI compiler is a one pass compiler.)

The above characteristics are desired for several reasons. First, it was desired that programs be compiled rather than interpreted for greater run time efficiency.

Second, the read only feature of the object code permits the executive to omit writing the code back to disc when each run time execution slot terminates.

Third, the immediate compilation allows several important additional advantages to accrue to the interactive user. Among these are:

A. The compiler can be used as a desk calculator.
B. Complete symbolic debugging aids (the principal advantage of interpreters) are still available due to the easy access of the compiler and symbol table at run time.
C. The user can symbolically alter variables in a running program without re-compiling or re-starting.
D. The user can cause each program statement to be executed (incremental execution) while it is being compiled a line at a time (incremental compilation).

The compiler itself is composed of two sections; a set of subroutines which are hand coded and the code generated by META PI. The subroutines fall into two classes.

a. Those which are not sensitive to the language being compiled. These routines are used by both FORTRAN PI and META PI and as such can be used for generating new compilers. An example of this class of subroutines if INUM; this subroutine tests the input stream for a digit string of arbitrary length.
b. A set of subroutines whose generality is a function either of the hardware on which the compiler is being implemented or the particular source language itself. For example, the routine EFFI detects the occurrence of certain instruction pairs and replaces this pair with a single instruction. This replacement is obviously dependent on a specific hardware instruction set. Another routine FLB is used to detect valid FORTRAN PI FORMAT statements. Such a routine is unique to FORTRAN and is not useful in the implementation of other languages. These non-transferable routines comprise less than 5% of the total FORTRAN PI object code.

The second section of FORTRAN PI is composed entirely of code generated by META PI. This coding performs a left to right scan of the source text, testing for syntactic units exactly as specified by the input to META PI. The structure of the input to META PI will be taken up shortly.

FORTRAN PI accepts source statements from terminal users and generates the machine code necessary to to carry out the intent of the statement. The compiler uses five regions in creating the users object program. These regions are created in ½ page blocks. A ½ page is 2048 memory locations (bytes); this is the minimum size block that can be memory protected on

the Spectra 70/45. The number of ½ pages allocated to each region is user specifiable at the time his program is created. The five regions are allocated as follows:

Regions 1 and 2

Contains the compiler's working storage, a specially constructed statement table and the source program label table.

The statement table is used for symbolic debugging; it enables the user to trace his program on selected criterion. For example, the program can be halted at any statement number, or the user can cause a symbolic printout when the value of specified variables change.

The label table contains information on every program variable and statement which contains a statement number.

Region 3

This area contains the user's compiled code. The generated code is "read only" and self-relocating. As a result the code ½ pages need never be written when the program is being staged out at the end of an execution time slot (the maximum time slot is ½ second).

Since the code generated is self-relocating it can be used as shared code on virtual memory hardware even though the current implementation is on a processor without virtual memory capabilities.

Region 4

This region contains constants that appear in the user's source statements and all variables that have been declared in COMMON statements.

Region 5

This area contains the value of variables not in COMMON, DO loop indices and the recursive function stack area.

FORTRAN PI functions and subroutines are recursive. The dynamic memory requirement needed to efficiently support a recursive process is obtained from Region 5. Thus the actual storage used in Region 5 expands and contracts dynamically during execution of the user's object program.

Since FORTRAN PI is implemented in the main by META PI its structure, which is designed for efficiency in the time sharing environment, is in fact determined by META PI. The implication is that any other language implemented using META PI would also have this region oriented structure; this without any special effort on the part of the language implementer using META PI.

The full implication of the nature of the compilers generated by META PI will be amplified when the implementation of Dartmouth BASIC using META PI is discussed in a later document.

META PI is a *problem* oriented language, that is, it is designed for use by individuals implementing entire compilers, syntax checkers, or for extending the capability of current compilers to satisfy special language requirements.

It has long been proposed that the structure of languages must be placed within the domain of the user; the logic is that only the user can be truly sensitive to his own specific needs. It is the purpose of problem oriented languages to achieve just this end, that is, they provide the user with a language that enables him to solve problems with special structural characteristics that would be either extremely difficult or, from the economic point of view, impossible to solve with procedure or assembly level language.

One of META PI's problem oriented objectives aims at providing the user the ability to create languages suited to his own needs without requiring that the user be familiar with specific computer hardware or the basic internal structure of compilers. This goal has yet to be achieved in its entirety, but META PI has demonstrated that the concept is feasible and that it's only a matter of time before the user will be provided with the capability for developing his own languages just as he is now able to create his own programs; the only remaining problems to be solved relate to the extent to which symbolics should be used within the compiler-compiler languages themselves.

In order to define a problem oriented language it is first necessary to examine the characteristics of the problem that the language is to slove

The language of a compiler-compiler (META PI) must be designed to solve the problem of compiler generation.

Compilers perform two basic functions:

1. They scan input statements in order to determine their validity within the definition of the language. The valid statements within a language is established by the *syntax* of that language. For example the Dartmouth BASIC statement

$$10\ \text{LET}\ X = X + 1$$

is valid
while

$$10\ \text{LET}\ X = \text{JOHN} + 1$$

is not, since in Case 2 the variable JOHN is not permitted in the language and hence is *syntactically* incorrect.

2. The second requirement of a compiler is the generation of the necessary computer intructions for effecting the execution of syntactically correct statements. This phase of the compiler implies that a meaning (semantics) is to be associated to a given statement. The meaning supplied takes the form of generated object code.

A compiler-compiler then must contain structural elements necessary to provide, in the compiler it produces, the ability both to scan for correct statements (syntactical structures) and also to produce object code. The user of such a language is freed of all the details that are involved in the actual generation of machine code required to implement the compiler itself.

META PI uses as its basic language structure the META series of compiler-compilers described by D. V. Schorre[2] and his associates at the UCLA computing facility. Its implementation, however, unlike the META series of compiler-compiler of the UCLA group is intended primarily for interactive software system. It has been used to generate two interactive compilers that are used on a production basis.

The basic parsing algorithm of the META type compiler is top-down left to-right, and deterministic. Briefly, "top-down" means the compiler first decides which rule should be satisfied next and then checks the input (or calls new rules) according to the alternatives of the rule. A "bottom-up" parser would, on the other hand, first check the nature of the input and then determine which rules could be used to describe it. A top-down, deterministic algorithm was selected for three principle reasons.

1. Coding can be generated immediately for the META statements as they are read in. This meshes with the goal of having incremental compilation.
2. Errors are easily pinpointed in deterministic parser. Backup is provided only when explicitly specified in the META PI language.
3. Deterministic parsers are faster than non-deterministic parsers.

As has been stated, the first requirement of a compiler-compiler language is to provide the language it creates a syntax checking capability. Fortunately, the syntactical description of programming languages has been provided a powerful symbolism in the Backus Normal Form (BNF).

BNF achieved its fame from its use in ALGOL '60 but is suited for describing a broad class of languages. It provides an excellent vehicle for the statement structure of a compiler-compiler. In order to enable the generated compiler to syntactically test the input statement, a BNF description is converted by META PI to generated code that will perform snytactic tests on the input statement. Though it is well suited to the syntactic phase of a compiler's work BNF was not designed with the intent of attaching semantic meaning to the statements involved.

It is in the area of semantics that the major effort in design has occurred in the development and definition of META PI.

META PI is computer program written for the RCA Spectra 70 that accepts the description of a language in extended Backus Normal Form. Both the syntactic and semantic functions of the compiler to be generated are contained within a single META PI statement. The output of the interactive version of META PI is (read only, sharable) Spectra 70 machine code which is the compiler for the language being described. This output code is unique to a given on-line user and does not interfere in any way with other on-line users who are sharing META PI interactively. The user of META PI can in fact have any number of different languages in various stages of development; the system does not distinguish between programs written in FORTRAN PI and those written in META PI; FORTRAN PI, META PI and the user's compiler form an integrated language system in RCA BTSS II.

The object code produced by META PI consists primarily of a set of subroutine calls which perform a recursive left to right scan of the source statements of the particular compiler language it describes.

META PI statements are designed to resemble Backus Normal Form. It was important, however, to extend BNF in order to include semantic operations (code generation) within the syntax structure describing the language and to simplify the description of the language. Four extensions were invloved:

1. The inclusion of factoring and the addition of an iterative operator. For example the BNF statement

$$A::= B/AC/AD$$

becomes

$$A:= B\$\,(C/D)$$

These changes were necessary for two reasons. First, the use of the $ sign enables the compiler to identify an iterative operation immediately on the appearance of the dollar sign ($). This greatly simplifies the compilation process. Second, since META PI is an interactive language the $ notation reduces input requirements thus increasing terminal efficiency. Furthermore, from the purely descriptive point of view, it simplifies the identification of proper strings defined by the statement, since the $ can be interpreted to mean "followed by

an arbitrary sequence of." Hence the sample META PI statement above is read as:
"An A is a B followed by an arbitrary sequence of C's or D's."
From just the visual examination of the string

BCCDDDCCCDDDDD

it is difficult to determine using the BNF descriptor whether or not the string is valid. With the extended BNF descriptor of META PI however it is immediately obvious that the above is in fact a valid string, that is a B followed by an arbitrary sequence of C's or D's.

2. The semantics are included within the syntax of a statement. This allows for object code to be generated as the scan of the source statement proceeds; in the vast majority of statements scanned, the complete generation of code and end of scan will occur simultaneously.
3. The ability to backup the code generation to some previous scan point is provided through special commands that are part of the META PI statement structure. This feature allows for efficient identification of those statement strings belonging to a language but not immediately identifiable on a left to right scan basis as a particular statement type. Consider for example the FORTRAN PI statement

DO1I = 1.5

This statement is a valid assignment statement which assigns the value 1.5 to the variable DO1I. If the syntax analysis, however, begins analyzing the statement as a DO statement it will not be rejected as such until the analysis of the statement is nearly completed. The backup facility of META PI provides an efficient means for re-evaluating the input string as a different statement type. It must be noted that the backup facility is provided for the scan of the source statement to the compiler being generated. It is *never* necessary to backup during the scan of a META PI statement since META PI is a deterministic language.
4. The compiler writer is provided with the capability of generating compile time error comments via a special error command which is also an integral part of the META PI statement structure. (This feature is not avaible in the interactive version of META PI described here.)

Before proceeding with a discussion of how META PI statements are written a discussion of META PI vs. BNF syntax is in order. The following conventions will hold:

META PI	*BNF*
: =	:: =
/	|
ABC	<ABC>
: ABC :	ABC

In addition:

1. A ; will terminate a META PI statement (unnecessary in the on-line version).
2. ( ) [parentheses] will be used to simplify BNF and will indicate factoring.
3. A $ replaces BNF finite state recursion.

To solidify META PI syntactical symbolism a few Dartmouth BASIC statements are shown below in BNF and META PI.

BASIC READ statement

BNF

<READ statement> :: = READ <read list>

META PI

READST : = : READ : READLST

BASIC read list

BNF

<read list> :: = <variable> | <read list>, <variable>

META PI

READLST : = VAR$ (:,:VAR)

BASIC FOR statement

BNF

<FOR statement> :: = FOR <simple variable> = <expression> TO <expression> <OPTEXP>

<OPTEX> :: = STEP <Expression> | <EMPTY>

META PI

FORST : = : FOR : SIMVAR : = : EXP : TO : EXP (:STEP:EXP/.EMPTY)

These examples are included to illustrate the similarities of BNF and META PI syntax. For the purpose of

these illustrations an effort has been made to name syntactic components to convey the same meaning they had in the BNF statement. For example the BNF < expression > became EXP.

It should again be emphasized that BNF does not include any facilities for including semantics operations within syntax operations hence none of META PI's semantic operations were shown.

META PI statements contain 3 types of elements:

1. Syntactic elements; these elements are compiled into code in the user's compiler that will test for syntactic elements in the source input to the user's compiler. These elements, then, are used to generate the "sieve" statement identifier or syntax checker of the user's compiler.
2. Semantic elements; the elements are compiled into code in the user's compiler that will effect the generation of object code.
3. META syntactic elements; these elements are compiled into code in the user's compiler that will enable it to efficiently resolve possible conflicts (ambiguities) in the newly defined input source statement via a backup facility. The user constructs META PI input statements by combining these three elements so as to produce his own compiler.

The general form for a META PI statement is:

LABEL: = expression

The left hand side is a unique identifier which serves as a reference to the expression on the right hand side (a META PI identifier is defined as a letter (A-Z) followed by an arbitrary sequence of letters or digits). For example the META PI statement which defines a digit would appear as:

DIGIT: = :0:/:1:/:2:/:3:/:4:/:5:/:6:/:7:/:8:/:9:

The name DIGIT can then be used on the right hand side of an expression to effect the test for a digit.

The character pair : = serves as a delimiter and distinguishes META PI statements from FORTRAN PI statements. The reader is reminded that META PI and FORTRAN PI are one integrated language package.

The expression is compiled into code in the user's compiler which is recursive, that is, the expression can contain a reference to itself either directly or indirectly.

When META PI generates the code for the expression within the user's compiler it will be generated such that it can have one of three results after being called.

1. True. This results if the input scanned as a result of being called satisfies the expression. The called routine will return with a truth indicator set, the input pointer will be moved past the data correctly scanned.
2. False. The input does not satisfy the expression, in this case the input pointer will be unaltered. The truth will be set indicating false.
3. Error. The expression prefix is correctly identified but the suffix is not. For example the statement

GO TO 20.3

is an invalid GO TO statement. The prefix GO TO is (possibly) correct but the suffix 20.3 is not. When this occurs an error routine is called, the input pointer is partially updated, the error routine will then insert a ? (question mark) after the last character successfully scanned.

These three conditions describe the behavior of the code that is generated in the user's compiler by a META PI expression. Some of the elements that comprise these expressions will now be discussed in detail.

Syntactic elements

:XXX.....X: The X's represent any character string. This syntactic element will create code in the user's compiler to test the current input for the string within the colons. In the DIGIT statement, shown previously, code would be generated that would test for a 0 or a 1 or 2 etc.

ABC This results in the generation of code in the user's compiler which will result in a call to the routine named (ABC in this case). This routine will presumably be written by the user with META PI. DIGIT defined above is such a routine; it could be used, for example to identify a number.

INUM: = DIGIT$DIGIT

The name could also designate one of the currently existing FORTRAN PI routines. This syntactic element is one of two possible methods available for linking to subroutines within META PI. The

second method involves preceding the routine name with a period. When the period notation is used META PI will assume that the routine called is not recursive and that a truth indicator is to be returned. When a routine is called without a period recursion is then possible by the routine called; a truth value will be returned by the called routines in either case.

.ID — This is the test for an identifier. Code is generated to link to the ID routine. Note the use of the period. The implication is that the ID routine does not subsequently link to itself.

.EMPTY — This is a special snytactic test which forces the true setting of the truth indicator.

.INT — This is a test for a FORTRAN integer.

.NUM — This is a test for a number which could (approximately) be defined by the following META PI statement:

```
NUM: = $DIGIT(:.:/.EMPTY)
$DIGIT(:E:(:+ :/:-:/.EMPTY)

DIGIT(DIGIT/.EMPTY)/
.EMPTY)
```

The code generated for this statement will identify numbers such as

1.23E-01

.001371E-15

1.361, 0123, 1E1

the definition could be read as:

"A NUM is equivalent to zero or more digits followed by an optional period followed by zero or more digits followed by the optional sequence; E followed by an optional plus or minus followed by a digit followed by an optional digit."

The NUM definition is relatively simple yet it illustrates factoring, iteration ($), tests for syntactic elements and the use of .EMPTY; a clear understanding of these elements will benefit the reader when other examples are given later in this document.

LKUP — This results in code being generated in the user's compiler that will link to the LKUP routine; this routine scans the label table for the last input detected. Label table entries are statement numbers or variable names. Each entry also contains appropriate control information such as type, memory address and program level. The routine will return one of three possible results.

1. The input was in the label table and assigned memory location is defined.
2. The input was not found in the label table.
3. The label was found but its memory location is yet to be defined. This type of entry is caused by forward references. For example a GOTO statement that specifies a statement number that has not yet been entered.

.TYPE(:NNYY:) — This routine looks up the input passed to it in the label table and tests if the type byte is in the class allowed by the argument NNYY. One function of this routine is to check for mixed mode errors.

.XXXX — Here the X's represent an arbitrary identifier. The use of this notation will cause META PI to generate linkage to the subroutine named by the symbol. The execution of the subroutine is assumed to effect a test on the input string. The results of this test will set the truth indicator which is returned to the calling routine. This notation is, in fact, the vehicle used by META PI in generating linkage to those syntactic routines previously discussed (.ID, .LKUP, etc.). In

addition to the symbols already defined, the user of META PI can link directly to those routines (written in META PI) that are used in creating the FORTRAN PI compiler; there are over 100 such routines most of which perform functions common to algebraic compilers. The META PI implementation of FORTRAN PI appears in Appendix 1.

Semantics—Code generation in META PI

The syntax operations permit the user who is implementing his own compiler to perform the statement identification function of the compiler being generated. The code generation that will effect the intent of a given source statement is handled by the semantic functions, these functions are imbedded in the META PI statement structure.

The semantic functions are composed of two sub elements:

1. Semantic commands.
2. Semantic operations.

Semantic operations are always contained within semantic commands. The general form is

sematic-command (semantic-Operations).

Semantic commands

Every semantic command has a direct effect on code generated by the compiler. When META PI encounters a semantic command in the input statement it will generate in the user's compiler the object code necessary to generate an element of an object program. There are five basic semantics commands.

.OUT(...) This command causes the current contents of the output area (a temporary area where code is being created by the user's compiler) to be converted to internal form and placed in the user's code area. The output area is a staging area for intermediate output that is in a semi-symbolic form. The code area contains the precise object code that will be executed by the computer. The output itself (the strings that are entered into the output area) is produced by the semantic operations that are specified within the parentheses (which are shown above as (....)). Three alternate actions can occur depending on the structure of the semantic operations contained within the .OUT(...) command.

1. If the first character is not a letter or a digit, then all subsequent characters are copied directly into the code area until a final colon (:) pair is detected.
2. If the fourth character is a period or a space it is assumed that the output is an instruction using an index register and with a symbolic address following the period or space located in position 4. The symbolic address will be looked up in the label table and from information contained there a real machine address will be generated.
3. If the above two cases fail, the character string is assumed to be machine code and it is converted directly into the code area.

.LABEL(...) This takes the current contents of the output area and places it into the label table. An error results if the label is already defined. The current value of the code area location counter will be associated with the label.

.IGN(...) This command will ignore (delete) the contents of the output area. This is useful since several semantic operations produce side effects, such as releasing registers, in addition to generating code.

.NOP(...) This command is used to produce the effect of the semantic operations without doing anything else. The results of the semantic operations will be left in the output area.

.DO(...) This is a specialized command whose effect is to cause META PI to execute immediately the instructions contained within the parentheses.

Semantic operations

The semantic operations are used to generate code in the output area. The code generated in the output area by these operations is in a symbolic form and not immediately executable; additionally these operations are generally constrained not to alter the input pointer or the truth indicator. A pointer is maintained to remember the next available location in the output area.

This pointer is updated after each semantic operation. These operations are listed below.

:CCC...C: Suffix the string between the colons to the output area. Note that no ambiguity exists with syntactic elements contained within colons since this notation has unique meaning depending on whether it has occurred inside or outside of a semantic command.

* Suffix the current input to the contents of the output area. This is generally used in conjunction with a successful .ID test. To emphasize the different roles being played by META PI, the user's compiler source statement which is input to the user's compiler, and the resulting object code, this simple operation will be explained further. When the * is found in a META PI string, code will be generated in the user's compiler to effect the placement of the last input into the output area. This code is part of the user's compiler. When a source statement is supplied to this compiler the user's compiler will effect symbolic code generation in the output area. This output area will then be converted into executable machine code.

S Save a copy of the current contents of the output area in a pushdown list and push the list.

R Restore (suffix to the output area) the top of the pushdown list and pop the list.

I Ignore (pop) the top element in the pushdown list.

X Swap the top two elements in the pushdown list.

*1 Generate a globally unique 4 byte character string beginning with the character #. This string will be locally constant and serves as a convenient way to label and reference locations in the generated code.

There are a set of semantics routines which facilitate the use of the general purpose and floating point registers of the Spectra 70 processor in the output code. A type of pushdown list for both of these register types is maintained at run time. There are 6 general purpose and 4 floating registers available to these semantic operations. If more registers are needed, coding will automatically be generated to implement saving and restoring of registers. This save and restore operation is a side effect of the following semantic routines.

OF Output the current general purpose register.

O Output the current floating point register.

\+ Output the next free general purpose register and make it current.

+2 Output the next free floating point register and make it current.

− Output two general purpose registers. The first one is the previous register, the second is the current register. When the operation completes the previous register will be made current. The output is always a digit pair. This format is specialized to take advantage of the register to register operations available on the Spectra 70 class of processors.

−2 Output a pair of floating point registers. The action is the same as the semantic operation for general purpose register pairs.

One final set of elements of a META PI statement have yet to be discussed namely the Meta Syntactic Commands. These commands are included primarily to permit efficient backup facilities in the user's compiler.

META syntactic commands

.LATCH(name) This causes code to be generated in the user's compiler that will result in the routine named in parentheses being called. In addition, if the routine (or any routine sub-

sequently called by the latched routine) exits to the error routine, backup will be affected.

C — This command can occur wherever a semantic operator can occur; it causes code to be generated in the user's compiler that will suppress the occurrence of a .LATCH in the calling routine. This command is generally used when initial ambiguity in a sub-expression has been resolved. Typical examples are when the first comma is detected in a FORTRAN DO statement, or when the logical operator is detected in a logical IF statement. Backup will not occur if a subsequent syntactic error is discovered and the error pointer will more clearly reflect the location of the error in the input statement.

.CLAMP — This command can occur wherever C can occur. It directs the compiler to suppress all preceding .LATCH's that are still in effect. .CLAMP is useful when .LATCH did not occur on the immediately preceding level, or when it is desired to inhibit the PI compiler or META PI from later attempting to scan input intended for the user's compiler. The reader is reminded that META PI, FORTRAN PI and the user's compiler are, in fact, part of an integrated language system.

The task now is to describe how the META PI elements are formed into statements which are used to create a user's compiler. The approach to be used in accomplishing this end will be via example. First several simple examples will be described. Then the entire META PI implementation of FORTRAN PI will be included as an appendix.

EXAMPLE 1.

FORTRAN PI allows the user to include comments in each statement after a concluding semicolon. If the user did not want this feature, but rather desired to permit multiple statements on one line (similar to ALGOL), he could write the following META PI command:

```
USERCC: = LABST.NOP(.CLAMP)$LABST
```

where LABST refers to the FORTRAN PI definition of a (possibly) labelled statement (see Appendix). The meta syntactic command .CLAMP disables the backup mechanism, and allows the error pointer to clearly reflect the location of a possible error in the subsequent arbitrary sequence of labelled statements ($LABST). Thus, the program segment

```
X = 1 + Y
Z = SIN(X) + W
Y = Y + 10
PRINT 1,X,Y,Z
```

could become

```
X = 1 + Y; Z = SIN(X) + W; Y = Y + 10; PRINT
                                1,X,Y,Z
```

EXAMPLE 2.

There is no efficient way to shift logically in the FORTRAN IV language. A FORTRAN PI user at RCA Laboratories required such a shift in order to improve the efficiency and readability of his program in which he made extensive use of bit manipulation. He used the following META PI statement:

```
USERCC : = :SHIFT: .NOP(.CLAMP) (:L: .SAV
          (:89:) /
          :R: .SAV(:88:)) .ID (INTV) :,: IEXP1
         .OUT(:58102000:R:103000:).OUT
          (:50102000 05E9,:—.E901)
```

This permitted him to enter statements like

SHIFTR J, 3	shift the variable J 3 bits to the right.
SHIFTL K, J + 5	shift the variable K J + 5 bits to the left.

Note the use of .SAV(...) to save the op-code of the shift instructions. INTV and IEXP1 are references to FORTRAN PI syntax. The "05E9" (BALR 14,9) constitutes a return to the executive and allows variable tracing (and other debugging aids). The fact that this is an assignment statement is communicated at compile time via the .E901 function.

EXAMPLE 3.

To further illustrate how META PI can be used to create new compilers, two statements from the imple-

mentation of Dartmouth BASIC language alluded to later will be discussed. The BASIC statements have been selected on the basis of their ability to convey the structure of META PI statements and not on the simplicity or complexity involved in their actual implementation.

The BASIC READ statement

This statement has the BNF format

< READ statement > : : = READ < read list >

In META PI the statement becomes

READ: = :READ:RID$(:,:RID):;:

META PI will scan the statement from left to right generating the following code:

1. A test for the word READ.
2. Linkage to the definition RID. This is a definition contained within the META PI definition of BASIC.
3. Instructions to effect iterative loop that will test for a comma followed by a read identifier.
4. A test for the line termination character ";". This character is appended to the statement internally.

This META PI definition is totally syntactic. The semantics for the READ statement are handled in the RID definition.

Handling relational operators

BASIC allows six relational operators; these operators are used within the BASIC IF statement; the operators permitted are:

BASIC operator	*Interpretation*
<>	not equal
<=	less than or equal to
>=	greater than or equal to
=	equal to
<	less than
>	greater than

The META PI definition for a relational is as follows:

REL = :< >: .SAV(:7:)/:< =: .SAV(:6:)/
:> =: .SAV(:A:)/:=: .SAV(:8:)/
:<: .SAV(:4:)/:>: .SAV(:2:)

META PI will generate the code equivalent to a sieve on the six possible relational operators. When one of the operators is detected a single character is entered into the pushdown list. This is effected by the .SAV semantics routine. This character is in fact the actual machine code representation of the branching condition. The REL definition is a sub definition of the IF statement. During the scan of the IF statement the character previously entered into the stack by REL will be popped into the output area and the complete branch instruction will be generated.

EXAMPLE 4.

The preceding example have shown how META PI provides a vehicle to allow user controlled generation of code which may be executed later at run time. The following example shows that the user can also control the generation of code to be executed at compile time, that is, he can generate a compiler-compiler. The example shows, first, the definition of a familiar language called BNF. Then in the new BNF language a simple syntax checker is defined. Then some test strings are entered.

```
/PRINT# BNF

   10 USERCC:=.NOP(.CLAMP)BNF:;:/:<:.ID.LABEL(*):>: ::: ::: :=:BX1:;:.O
      UT(:07FA:)

   20 BX1:=BX2$(:!:.OUT(:58E.:*1).OUT(:078E:)BX2).LABEL(*1)

   30 BX2:=BX3.OUT(:58E.:*1).OUT(:077E:)$(BX3.OUT(:477.ERR:)).LABEL(*1)

   40 BX3:=:<:(:EMPTY:.OUT(:0420:)/.ID.OUT(:41E.:*).OUT(:450.LATC:)):>:
      /STRING.OUT(:45E.TEST:).OUT(:::#R:::)

   50 STRING := ALPHABET .SAV(*) $(ALPHABET .SAV(R*))
```

```
   60 ALPHABET := LETTER /DIGIT/ :?:/:":/:$:/:#:/:%:/ETC
   70 LETTER := :A:/:B:/:C:/:D:/:E:/:F:/:G:/:H:/:I:/:J:/:K:/:L:/ETC
   80 DIGIT := :0:/:1:/:2:/:3:/:4:/:5:/:6:/:7:/:8:/:9:
   90 <BNF> ::= 0 ! 1<BNF>1
  100 101
  110 11011
  120 1111111111111111111110111111111111111111
```

```
/CODE
   130 11011!
   130 ERROR  11011? 1;
   130 111011
   130 ERROR  11011? ;
   130
```

```
/PRINT#
   10 USERCC := .NOP(.CLAMP) BNF :;:/:<:.ID.LABEL(*):>: ::: ::: ::= BX1
      :;:.OUT(:07FA:)
   20 BX1 := BX2$(:I:.OUT(:58E.:*1).OUT(:078E:)BX2).LABEL(*1)
   30 BX2 := BX3.OUT(:58E.:*1).OUT(:077E:)$(BX3.OUT(:477.ERR:))
      .LABEL(*1)
   40 BX3 := :<:(:EMPTY:.OUT(:0420:)/.ID.OUT(:41E.:*).OUT(:450.LATCH:))
      :>:/STRING.OUT(:45E.TEST:).OUT(:::#R:::)
   50 STRING := ALPHABET .SAV(*) $(ALPHABET .SAV(R*))
   60 ALPHABET := LETTER / DIGIT / :?:/:":/:#:/:$:/:%:/:&:/:':/:(:/:):/
      :*:/ETC
   70 LETTER := :A:/:B:/:C:/:D:/:E:/:F:/:H:/:I:/:J:/ETC
   80 DIGIT := :0:/:1:/:2:/:3:/:4:/:5:/:6:/:7:/:8:/:9:
   90 <BNF> ::= 0 ! 1<BNF>1
  100 101
  110 11011
  120 1111111111111111111101111111111111111111

/CODE
  130 111011
  130 ERROR  11011? ;
  130 111011111
  130 ERROR  10111? 11;
  130
```

After the FORTRAN PI compiler was successfully implemented the challenge to implement a different language using META PI was irresistible for two reasons:

1. Since META PI and FORTRAN PI evolved simultaneously there was some question as to whether or not the generality of META PI had been seriously affected by efforts to accommodate the peculiarities of FORTRAN.
2. Evidence had to be accumulated that would tend to demonstrate the leverage that can be gained by using the compiler-compiler approach.

It was decided to use Dartmouth BASIC as a test case for gathering data to support conclusions for the above hypotheses. The results of the implementation of BASIC with META PI were startling, even to the author.

With no pre-preparation of any kind the project to implement BASIC began on April 15, 1968. On May 6, 1968 BASIC was available to users of BTSS II on an interactive basis; total elapsed time to implement the language was 3 weeks.

The entire implementation was done interactively. The implementation hours for man, console and processor time are as follows:

* Total Man Hours—96
* Total Console Hours—33
  This is the time the Teletype was connected to the computer.
* Total Processor Hours — Less than .1 hour
  This is the time the CPU was performing functions for the implementation of BASIC.

The results have in the author's mind further strengthened the conviction that availability of interactive on-line compiler-compilers such as META PI can increase language implementation efficiency by an order of magnitude and provide computers which the much needed capability of creating and modifying languages to suit individual user's own special needs.

ACKNOWLEDGMENTS

The author wishes to express deep appreciation to M. Pelligrini for his assistance in preparing this document and the examples, H. A. Freedman for his collaboration and implementation of the Executive routine, and N. L. Gordon and John Carr III for suggesting and guiding the project.

REFERENCES

1 P NAUR Ed
Report of the algorithmic language ALGOL 60
CACM Vol 3 265 May 1960 pp 299–314

2 D V SCHORRE
META–II a syntax-oriented compiler writing language
Proc ACM 19th National Conference 1964 p D1 3

3 M E WHITE
A remote processing system for the Apt language
Proc AFIPS current conference
The reference list is abbreviated since both an excellent list of references and an excellent exposition is contained in:

4 J FELDMAN D GRIES
Translater writing systems
CACM Vol 11 No 2 Feb 1968 pp 77–113

APPENDIX

The following pages list the actual input to an off-line version of the compiler-compiler. The off-line version has several minor differences from the on-line compiler-compiler, and one major difference. The major difference is that the off-line output is a symbolic Spectra 70 assembly input tape from which the FORTRAN PI compiler is assembled, rather than the direct machine code generated on-line.

Some other differences are:

1. "=" as statement delimiter rather than ":=".
2. Absence of "." as delimiter before some semantic operations (E2, E3, E986, E987, E1, E900, E901, . . . , E908).

3. .EL is a pseudo semantic operation. It actually performs a test for a ; and "return"s or generates the "SHOULD END HERE" message.
4. .ERR(:....:), when used, denotes the actual error message to be displayed if the preceding test fails.
5. EFF OFF and EFF ON are special commands to the off-line compiler tell it to turn off and on some special internal optimizing code.
6. The X semantic operation here is identical to the on-line Z.
7. There are several subroutines referenced but not defined. This is usually because they have been partially hand coded. At any rate, the explanation of all the features of the off-line compiler-compiler is beyond the scope of this document.

Any of these FORTRAN PI routines can be accessed by the user (via his own compiler).

A-1

```
INTV=(.IID.OUT(:41:+: :*)/.IPR.OUT(:58:+: :*))SUBEXP;
RLV=(.FID.OUT(:41:+: :*)/.SPR.OUT(:58:+: :*))SUBEXP;
RLDV=(.TYPE(:EF80:).OUT(:41:+: :*)/.TYPE(:EF82:).OUT(:58:+: :*))DSUE ;
CMPXV=(.TYPE(:EFAO:).OUT(:41:+: :*)/.TYPE(:EFA2:).OUT(:58:+: :*))CSUBXP;
IEXP1=IEXP2$(:+:ITERM.OUT(:1A:-E2)/:-:ITERM.OUT(:1B:-E2));
SEXP1=SEXP2$(:+:STERM.OUT(:3A:-2E2)/:-:STERM.OUT(:3B:-2E2));
DEXP1=DEXP2$(:+:DTERM.OUT(:2A:-2E2)/:-:DTERM.OUT(:2B:-2E2));
CEXP1=CEXP2$((:+:.SAV(:A:S)/:-:.SAV(:B:S))CTERM.OUT(:2:R-4).OUT(:2:R-4));
BEXP1=BTERM$(:+:BTERM.OUT(:16:-E2));
SUB2ST=.OUT(:47F09ED40080:R:00:)(:(:SUBRSC.OUT(:947F100:R):):/.EMPTY
      .OUT(:C000:)).NOP(.EL);
EQST=.LATCH(DOST)/EQ2ST;
NEQST=DECST/LABST/ENDST/SUBST/FCARD;
EQ2ST=.LATCH(LIFST)/.RLATCH(IOST)/ASST;
LABST=GOST/.LATCH(IFST)/:RETURN;:.OUT(:47F09E98:E908)/ENDOST;
DOST=:DO:.SAV(*1).INT.SAV(*: :).ID.IID.SAV(*).SAV(E987S).OUT(:41:+: !*):=:IEXP1
   .OUT(:50302000:):,:.IGN(C-)IEXP1.ERR(:NOT INTEGER:)(:,:IEXP1.ERR(
   :NOT INTEGER:)/.EMPTY.OUT(:41:+:00001:)).OUT(:411:RE986).OUT(:45E09E22 :
        ---E901).LABEL(*1).NOP(.EL);
LIFST=:IF(:(.LATCH(LSUBIX).OUT(:19:-E2).IGN(-)/
  DEXP1(RELOP.NOP(C))DEXP1.ERR(:NOT AN EXP:).OUT(:29:-2E2).IGN(-2)):):
   .ERR(:MISSING ):).OUT(.58E :*1).OUT(:47:R:09034:)L2ST.ERR(:NOT A STATEMENT:
    ).LABEL(*1);
GOST=:GOTO:(.INT.OUT(:58E :*).OUT(:47F0904C:F902)/:(:.OUT(:58E :*1).OUT(:05:+:
  E:).OUT(:4120000141F0904C:)GOINT.ERR(:NOT AN INTEGER:)$(:,:GOINT.ERR
  (:NOT AN INTEGER:))):):.ERR(:MISSING ):),OUT(:45E09018:).LABEL(*1).OPT(:,:)
    IEXP1.OUT(:07F2 :--E902)).NOP(.EL);
ENDST=:END;:.OUT(:47F09038:E900)ENDSB.ERR(:UNTERMINATED DO LOOP:);
ENDOST=:CONTINUE;:.OUT(:05E9:E900)/CALLST/LIFST/CST/IOST/:.:
   (:OUT(:.STRING.OUT(*E900)/:LABEL(:.STRING.LABEL(*))::):);
ST=.NOP(C)(.LATCH(CCST)/.DOEND.INT.LABEL(*): :(.EQUALS(EQ2ST)/
   ENDOST).ERR(:INVALID DO END:)
    DOGEN/: :(.EQUALS(EQST)/NEQST)/:B:BOOLST/.INT.LABEL(*): :.ERR(:BAD LABEL:)
  (.EQUALS(EQST)/LABST)/FCARD/.EMPTY: :.ERR(:BAD LABEL:)).IGN().IGN().IGN();
FCARD=(:F :/:EXTERNAL:)
        .ID.ERR(:BAD LABEL:)CALLSB$(:,:.ID.ERR(:BAD LABEL:)CALLSB).NOP(.EL);
L2ST=.EQUALS(EQ2ST)/LABST;
IEXP2=:-:ITERM.OUT(:13:0F0F)/:+:ITERM/ITERM;
SEXP2=:-:STERM.OUT(:33:00)/:+:STERM/STERM;
DEXP2=:+:DTERM/:-:DTERM.OUT(:33:00)/DTERM;
BTERM=BPRIM$(:*:BPRIM.OUT(:14:-E2));
ITERM=IPRIM$(:*:IPRIM.OUT(:181:0FE2).IGN(-).OUT(:1C0:0F:18:0F:1:)/
    :/:.SAV(0F)IPRIM.OUT(:180:R:8E0000201D0:0F).IGN(-).OUT(:18:0F:1:));
STERM=SPRIM$(:*:SPRIM.OUT(:3C:-2E2)/:/:SPRIM.OUT(:3D:-2E2));
DTERM=DPRIM$(:*:DPRIM.OUT(:2C:-2E2)/:/:DPRIM.OUT(:2D:-2E2));
CEXP2=:+:CTERM/:-:CTERM.OUT(:33:00:33:XX)/CTERM;
CTERM=CPRIM$((:*:.SAV(XX)CPRIM/:/:.SAV(XX)CPRIM.OUT(:45E09BDC0:XX:0:))
     .OUT(:45E09BB00:R:0 :-2).IGN(-2));
SUBEXP=SUBSCL.OUT(:1E:0F:1:)/.EMPTY;
DSUBXP=SUBSCL.OUT(:1E111E:0F:1:)/.EMPTY;
CSUBXP=SUBSCL.OUT(:1E111E111E:0F:1:)/.EMPTY;
ASST=.ID(INTVL:=:.ERR(:EXPECTED = HERE:)(.LATCH(IEXP1).OUT(:50302000 :-)/
   (.RLATCH(SEXP1)/.RLATCH(DEXP1)/CEXP1.IGN(-2)).OUT(:45E09E6050002000 :-2))/
   RLVL:=:.ERR(:EXPECTED = HERE:)(RHIEXP/
    .LATCH(SEXP1)/.RLATCH(DEXP1)/CEXP1.IGN(-2))
   .ERR(:NOT AN EXPRESSION:).OUT(:70002000 :-2)/RLDVL:=:.ERR(:EXPECTED = HERE:)
      (RHIEXP/.LATCH(DEXP1)/CEXP1.IGN(-2))
   .ERR(:NOT AN EXPRESSION:).OUT(:60002000 :-2)/CMPXVL:=:.ERR(:EXPECTED = HERE:)
     (RHIEXP.OUT(:2F:+20)/CEXP1).ERR(:NOT AN EXPRESSION:)
     .OUT(:6020200860002000 :-2-2)).OUT(:05E9 :-E901).NOP(.EL);
```

A-2

```
RHIEXP=.LATCH(IEXP1).OUT(:180345E09E00 :-+2);
IFST=:IF(:(.LATCH(IEXP1).OUT(:1222 :-)/(.RLATCH(SEXP1)/DEXP1)
              .ERR(:BAD EXPRESSION:).OUT(:3200 :-2))IFEND.ERR(:MISSING ););
BOOLST=.DUEND.INT,LABEL(*): :BASST(DOGEN)/(,INT.LABEL(*): :/: :)(,EQUALS
      (BASST)/BIFST);
DOGEN=$(.OUT(:411:RE986).OUT(:412 :RI).OUT(:58E :R).OUT(:45F09E9C:E903).MOREDO);
CCST=(: :/.EMPTY)(.LATCH(UCC)/.ID,LABEL(*)::=:.NOP(C)CCX2$(:/:.OUT(:078A:)
      CCX2):;:.OUT(:07FA:));
CST=(:FLOWON:.SAV(:028:)/:FLOWOFF:.SAV(:02C:)/:STOP:.SAV(:038:)/:PAUSE:.OUT(
    :92108000:).SAV(:024:)/ICEST)
      .OUT(:45E09:RE900).NOP(.EL)/NICEST;
SUBST=(:SUBROUTINE:/:SUBR,:).ID.ERR(:INVALID NAME:).OUT(:47F09038:E900)SUBRSB(
       SUB2ST).OUT(:05E9:);
IOST=(ALGIOST/
     (((:READ(:.SAV(:018:)/:WRITE(:.SAV(:000:))IEXP1.ERR(:NOT AN INTEGER:):,:
   .INT(FLB).ERR(:BAD FORMAT LABEL:).SAV(:581 :*):):.ERR(:MISSING ):)/
     (:PRINT:.SAV(:000:)/:READ:.SAV(:018:))
  (.INT(FLB).SAV(:581 :*)/.EMPTY.SAV(:4110B02C:)).OUT(:1F2:+)).OUT(R).OUT(:58F0B
   :R).OUT(:05EF :-).OPT(:,:)(IOSEQ$(:,:IOSEQ)/.EMPTY)
     .OUT(:45E0F00C:)/
   (:REWIND:.SAV(:008:)/:BACKSPACE:.SAV(:00C:))IEXP1.ERR(:NOT AN INTEGER:)
                .OUT(:58F0B00045E0F:R: :-))).OUT(:05E9:E900).NOP(.EL);
GOINT=.INT.OUT(:1F32:).OUT(:58E :*).OUT(:072F:);
LSUBIX=IEXP1(RELOP)IEXP1;
RELOP=:.:(:LE:.SAV(:3:)/:EQ:.SAV(:7:)/:NE:.SAV(:9:)/:GT:.SAV(:D:)/:GE:.SAV(:5:)/
    :LT:.SAV(:B:)).ERR(:BAD OPERATOR:):.:.ERR(:SHOULD BE A ,:)/:<:.SAV(:B:)
    /:=:.SAV(:7:)/:>:.SAV(:D:);
BASST=.ID(INTVL/RLVL):=:.ERR(:SHOULD BE =:)BEXP1.ERR(:NOT BOOLEAN:).OUT(:5030
     200005E9 :--E901).NOP(.EL);
SUBSCL=:(:GINDEX.ERR(:NOT AN ARRAY:)IEXP1.ERR(:NOT INTEGER EXPRESSION:)
                    (MIEXPS).OUT(:180:0F:45E0E004 :-):):.ERR(:MISSING ):);
BPRIM=.CHCON.OUT(:58:+.C4GEN)/(:.FALSE.:/:0:).OUT(:1F:+0F)/
     .BCONST.OUT(:58:+.BCGEN)/:-:BPRIM.OUT(:57:0F:0801C:)/
    (:.TRUE.:/:1:).OUT(:48:+:0801C:)/:(:BEXP1:):.ERR(:MISSING ):)/
     .ID(INTV/RLV).OUT(:58:0F:0:0F:000:E1);
CALLST=:CALL:(:CHAIN(:IEXP1:):.ERR(:MISSING );).OUT(:0AD1 :-)/
           .IOCALLSB.SAV(*)(:(:PLIST:):.ERR(:EXPECTED ):)
     /PLIST).OUT(:58F :R)).OUT(:05EF:E904).NOP(.EL);
ICEST=(:TRACE:(:ON:.SAV(:044:)/:OFF:.SAV(:048:))/:DUMP:.SAV(:03C:)
   /:PDUMP:.SAV(:040:))PLIST;
IPRIM=IPRI;
SPRIM=SPRI$(:**:(.LATCH(IPRI).OUT(:180:0FE2).IGN(-).OUT(:45E09FBE0:00:1:)/
     .OUT(:45E09FD40:00:1:)SPRI.OUT(:3C:-2:45E09FCC0:00:1:)));
DPRIM=DPRI$(:**:(.LATCH(IPRI).OUT(:180:0FE2).IGN(-).OUT(:45E09FBE0:00:0:)/
     .OUT(:45E09FD40:00:0:)DPRI
         .ERR(:ILLEGAL EXPONENT:).OUT(:2C:-2:45E09FCC0:00:0:)));
CPRIM=CPRI$(:**:.OUT(:45E09CB00:XX:0:).SAV(XX)CPRI.OUT(:45E09BB00:R:0 :-2).IGN
   (-2).OUT(:45E09C660:XX:0 :));
IOSEQ=:(:.OUT(:58E :*1).OUT(:05:+:E18:+:1:)$.LATCH(PARCOM).ID.IID,OUT(:1F :*)
   .SAV(E987S).OUT(:411:RE986).OUT(:45E09EFC181:0F: :-)
   .OUT(:07A:0F:07F1:).LABEL(*1).SAV(*):=:.ERR(:EXPECTED = HERE:).NOP(.CLAMP)
   IEXP1.ERR(:BAD EXPRESSION:).OUT(:50:0F: :R).IGN(-):,:.ERR(:MISSING ,:)
   IEXP1.ERR(:BAD EXPRESSION:)(:,:IEXP1.ERR(:BAD EXPRESSION:)/.EMPTY
   .OUT(:41:+:00001:)).OUT(:411:RE986).OUT(:180:0F: :-).OUT(:18F:0F: :-)
    .OUT(:90F01000051:0F: :-):):.ERR(:MISSING ):)
         /.SAV(:0:S)IOPARAM.OUT(:45E09E86:);
PARCOM=IOSEQ:,:;
INTVL=.TYPE(:FF45:).LEVL.ERR(:LEFT SIDE IS FUNCTION:).OUT(:41:+:0D030:)/INTV;
 RLVL=.TYPE(:FFC5:).LEVL.ERR(:LEFT SIDE IS FUNCTION:).OUT(:41:+:0D030:)/RLV;
RLDVL=.TYPE(:FF85:).LEVL.ERR(:LEFT SIDE IS FUNCTION:).OUT(:41:+:0D030:) RLDV;
```

```
CMPXVL=.TYPE(:FFA5:).LEVL.ERR(:LEFT SIDE IS FUNCTION:).OUT(:41:+:0D030:)/CMPXV;
BIFST=:IF(:BEXP1.ERR(:NOT BOOLEAN:).OUT(:1222 :-)IFEND.ERR(:SHOULD BE A ););
IFEND=:):,INT.ERR(:NOT AN INTEGER:).OUT(:41F0904C:).OUT(:58E :*).OUT(:074F:)
   :,:.ERR(:MISSING ,:).INT.ERR(:NOT AN INTEGER:).OUT(:58E :*).OUT(:07CF:E3)
   :,:.ERR(:MISSING ,:).INT.ERR(:NOT AN INTEGER:).OUT(:58E :*).OUT(:07FF:E3
    E902).NOP(.EL);
CCX2=$CCO(CCX3).OUT(:58E.:*1).OUT(:077E:)$(CCO/CCX3.OUT(:477.ERR:))
      .LABEL(*1E90);
NICEST=(:EXECUTE:.DO(:SR  8,8:)/:SAVE:
 (.DO(:LA 8,36:):SOURCE:/.DO(:LA 8,8:):OBJECT:/.DO(:LA 8,44:).OPT(.EMPTY))/
         :SQUEEZE:.DO(:LA 8,16:)/:CALC:.OPT(:ULATOR:).DO(:LA 8,28:)
          .OPT(.EMPTY)).OPT(ONOFF).OPT(.EMPTY)
          /:BATCH:.DO(:LA 8,24:));:.ERR(:SHOULD END HERE:).DO(:L 1,SAVSTK:)
         .DO(:EX 0,*+8(8):).DO(:B *+56:).DO(:USING STACK,1:)
         .DO(:OI SW,1:).DO(:NI SW,254:).DO(:NI SW,253:).DO(:OI SW,2:)
         .DO(:NI CLINE,251:).DO(:OI CLINE,4:).DO(:OI CLINE,128:)
         .DO(:OI SW,131:).DO(:NI SW,124:)
         .DO(:NI SW,127:).DO(:OI SW,128:).DO(:NI SW,125:).DO(:OI SW,130:)
                                    .DO(:DROP 1:).DO(:SPM 2:);
ALGIOST=(:READ(<:.SAV(:0:)/:PRINT(<:.SAV(:4:)).OUT(:580 :*1)
   .OUT(:50000068500006C:).OUT(:58F0802:R)PWORDC$(:,<:PWORDC)
 .OUT(:1FEE50E0C06::858E0C06C07F9:E900).LABEL(*1):):.ERR(:MISSING ):).NOP(.EL);
         ORG   ALGIOST+14 FOR TABLE GENERATION ONLY
CPRI=.LATCH(CELEMF)CEXP1:):.OUT(:45E09:R:0:XX:0:)/
          XCON(:,:XCON.ERR(:NOT A NUMBER:)/.EMPTY,OUT(:2F:+20))/
   .LATCH(XCONST)/:(:CEXP1:):.ERR(:MISSING ):)/
      .ID(CMPXV.OUT(:68:+2:0:0F:00068:+2:0:0F:008 :-)/.TYPE(:F5A5:)FCN
      .SAV(RS).OUT(:68:+2:0:R:03068:+2:0:R:038:)/DPRID.OUT(:2F:+20));
DPRI=.LATCH(ELEMF)DEXP1:):.ERR(:MISSING ):).OUT(:45E09F:R00:0:)/.LATCH(ABSF)
    DEXP1:):.ERR(:MISSING ):).OUT(:30:00)/XCON/:(:DEXP1:):.ERR(:MISSING ):)/
      .ID(DPRID).ERR(:BAD TYPE:);
DPRID=RLDV.OUT(:68:+2:0:0F:000:E1).IGN(-)/.TYPE(:F585:)FCN
      .OUT(:68:+2:0:R:030:)/
     RLV.SAV(:78:+2:0:0F:000:E1).OUT(:2F:00).OUT(R: :-)/.TYPE(:F5C5:)(FCN)
     .OUT(:2F:+20:78:0:0:R:030:)/ITORD;
SPRI=.LATCH(ELEMF)SEXP1:):.ERR(:MISSING ):).OUT(:45E09F:R00:1:)/
     .CHCON.OUT(:78:+2.C4GEN)/.BCONST.OUT(:78:+2.BCGEN)/
       '.NUM.OUT(:78:+2.NGEN)/:(:SEXP1:):.ERR(:MISSING ):)/
              .LATCH(ABSF)SEXP1:):.OUT(:30:00)/.ID(SPRID);
IPRI=.INT.OUT(:58:+.IGEN)/.CHCON.OUT(:58:+.C4GEN)/
      .BCONST.OUT(:58:+.BCGEN)/:(:IEXP1:):.ERR(:MISSING ):)
     /.LATCH(ABSF)IEXP1:):.ERR(:MISSING ):).OUT(:10:0F0F)
     /.ID(INTV.OUT(:58:0F:0:0F:000:E1)/.TYPE(:F545:)FCN.OUT(:58:+:0:R:030:));
MIEXPS=:,:IEXP1.ERR(:NOT INTEGER EXPRESSION:)
               (MIEXPS).OUT(:180:0F:05EE :-)/.EMPTY.OUT(:41E09F4E1F11:R);
SPRID=RLV.OUT(:78:+2:0:0F:000:E1).IGN(-)/.TYPE(:F5C5:)FCN.OUT(:78:+2:0:R:030:)/
      ITORD;
ITORD=(INTV.OUT(:5800:0F:000 :E1-)/
         .TYPE(:F545:)(FCN).OUT(:5800:R:030:)).OUT(:45E #ITOR:+2);
FCN=.SAV(*):(:PLIST:):.ERR(:MISSING ):).OUT(:58F :R).OUT(:05EF:E904).SAV(:1:)
              /.EMPTY.LEVL.ERR(:BAD FUNCTION CALL:).SAV(I:D:);
PLIST=.OUT(:58:+:0D000:).SAV(0F).OUT(:41:+:0:R:03C:)
    (PARAM2.SAV(:0:)$(:,:PARAM1.ERR(:NOT A PARAMETER:).SAV(I:4:)(:,:PARAM2
         .ERR(:NOT A PARAMETER:).SAV(I:0:)/.EMPTY)).OUT(:9680:0F:00:R)/.EMPTY
       .OUT(:92C0:0F:008:)).IGN(-).IGN(-);
ABSF=:ABS:.OPI(:F:):(:;
ELEMF=(:SQRT:.SAV(:DC0:)/:SIN:.SAV(:E40:)/:COS:.SAV(:EC0:)/(:LOG:/:ALOG:).SAV
   (:D40:)/:EXP:.SAV(:CC0:)/:ATAN:.SAV(:C60:)/:TANH:.SAV(:F40:)).OPT(:F:):(:;
XCONST=:(:DEXP1:,:.NOP(C)DEXP1.ERR(:NOT AN EXPRESSION:):):.ERR(:MISSING ):);
XCON=.DNUM.OUT(:68:+2.DGEN)/.CH8CON.OUT(:68:+2.C8GEN)/.B8CONST
```

```
      .OUT(:68:+2,B8GEN)/:.PI:.OUT(:68:+2:09DF0:);
PARAM1=.SAV(:4:S)PARAM;
PARAM2=.SAV(:8:S)PARAM.OUT(:41:0F:0:0F:008:);
CELEMF=(:SQRT:.SAV(:C90:)/:SIN:.SAV(:CD0:)/:COS:.SAV(:CD8:)/(:LOG:/:ALOG:)
     .SAV(:CB0:)/(:MAG:/:ABS:).SAV(:C14:)/:ARG:.SAV(:C2C:)/
    :EXP:.SAV(:C66:)/:ATAN:.SAV(:D3A:)/:TANH:.SAV(:D02:)),OPT(:F:):(:;
ONOFF=:ON:/:OFF:.DO(:LA 8,4(0,8):);
CCO=(:.OUT(:/:.IGN(:.OUT(:92FF900A:))$CCO1:):.OUT(:05E9:)/
   :.LABEL(:$CCO1:):.OUT(:45E.LABE:)/:.DO(:$(.SR.OUT(*):::):):/:.OPT(:CCX1:):/
     :.SAV(:$CCO1:):.OUT(:45E.SAV:)/:.NOP(:$CCO1:);
CCO1=CCOSUB.OUT(:45E.:*)/:C:.OUT(:92015000:)/.SR.OUT(:05E4:).OUT(:#:#*:::):::;
EFF OFF
CCOSUB=:*1:/:R:/:I:/:+2:/:+:/:S:/:-2:/:-4:/:-:/:X:/:OF:/:0:/:*:/:#:/:.:.ID;
EFF ON
CCX3=.ID.OUT(:41E.:*).OUT(:0503:)/.SR.OUT(:45E.TEST:).OUT(:#:#*:::):::/
    :(:CCX1:):/:.EMPTY:.OUT(:0420:)/:$:.LABEL(*1)CCX3.OUT(:58E.:*1).OUT(:078E:)
    .OUT(:0420:)/:.LATCH(:.ID.OUT(:41E.:*).OUT(:450.LATC:):):/
    :.TYPE(:.SR.OUT(:45E.TYP:).OUT(*):::):/
    :.:.ID.OUT(:45E.:*);
CCX1=CCX2$(:/:.OUT(:58E.:*1).OUT(:078E:)CCX2),LABEL(*1);
FCNST=:FUNCTION:.ID.ERR(:INVALID NAME:),OUT(:47F09038:E900)FCNSB(SUB2ST)
     .OUT(:45E09EBE:.X2);
SUBV=.TYPE(:1505:).OUT(:58:+: :*);
```