META USER GUIDE

Marinchip Systems Mill Valley, CA 94941

[illegible]

[illegible]

Marinchip Systems

[illegible] reason

16 St. Jude Road
Mill Valley, CA 94941 ■ (415) 383-1545

[illegible]

1. Credits

META is a product of Marinchip Systems, 16 St. Jude Rd., Mill Valley, CA 94941. This manual is not intended as a product specification. The description of META given in the file META.MET on the release diskette shall in all events be considered the final arbiter on how META works.

The purpose of this document is to explain the use of a syntax-directed compiler compiler in enough detail that the actual definition of the language may be read and understood.

2. META - A Syntax-Directed Compiler Writing Language

2.1. What does "syntax-directed mean?

Webster defines SYNTAX as:

1) A connected or orderly system: harmonious arrangement of parts or elements.
2) The way in which words are put together to form phrases, clauses, or sentences.

For our purposes, syntax means the underlying structure of a language that specifies how the smallest items ("tokens") are combined to make up statements and programs.

A syntax-directed compiler is one that processes the input source program against a description of valid syntax for the language, and generates code to perform the desired functions when the syntax pattern matches the input source messages when the input source code does not conform to the syntax description.

META is a language with which you describe the syntax of a target language - that language you wish to compile, and the assembly code that should be generated for each part of the source code that matches the syntax description.

2.2. The Use of a Compiler

In practice, a user will create a text source file using EDIT that contains the source code to be compiled. The Compiler will read this source file and create a file of assembly language statements that perform the desired functions. Control is then passed to ASM, which reads the intermediate assembly language text fil[illegible] writes a relocatable output text file, which describes the assembly statements in a numeric form, as if the program started at address 0000 in memory.

The user will compile all modules (main program and any subroutines) using the above process, and then will use the LINK program to make an executable binary file that contains the final, useable program. Each time the program is to be run, the name of the executable file is entered as a command to the operating system.

The process may be pictured as:

Keyboard input	>>	EDIT	>>	Source file
Source file	>>	COMPILER	>>	Assembly Code File
Assembly Code	>>	ASM	>>	Relocatable File
Relocatable Files	>>	LINK	>>	Executable Program

In practice, the compiler automatically executes the assembler, so the ASM step is transparent to the user. The user follows the pattern:

EDIT >> COMPILE >> LINK >> RUN

2.3. Writing a Compiler Using Meta

To write a compiler using META, you will need a very good understanding of assembly language programing, the function of compilers, and the ability to keep seperate time-related events coordinated. As a package, a compiler includes actions taken during the generation of the compiler, during the execution of the compiler, and during the execution of the compiled program. In describing some part of a compiler, you may set a META flag to allow some option to be

compiler while it is examining the source code it is to compile, and the run-time library may need set-up directions from your compiled code. Keeping these related but seperately timed events coordinated is perhaps the hardest part of compiler writing.

The task of writing a compiler may be broken down into the following steps:

1) You must describe the exact syntax of the language you wish to compile.

2) You must determine what assembly language code is to be generated in response to the various syntax elements.

3) You must write any run-time subroutines that will be needed by the compiled code.

4) You must debug and thus validate your compiler and run-time routines. This will actually consume most of your effort.

5) You must document your compiler and routines at [illegible] levels: The user's manual, and a program logic manual, so that someone else may maintain the compiler. It may be you six months later that will need explainations of why something was done the way it was.

This manual will attempt to introduce you to META, and explain in general how to use it. Only actual work with META and examination of it's output will make the pieces fall into place. While the use of META will not come easily, it is a very powerful tool that will let you successfully write compilers in a reasonable amount of time, and it is well worth the effort to learn.

2.4. The Nature of Syntax Descriptions

It is impossible to describe anything as complex as a language in a single definition. Thus the language is broken into several pieces, and seperate descriptions are given for each piece, and then a "master description" is made that shows how the pieces fit together. The more complex the language, the more levels of description that might be used.

One approach that might be used is to start our definitions with the smallest pieces and build up from there. Another is to start with the overall program and break it down into smaller and smaller pieces. Whichever approach you take depends on personal preference.

In this manual, the bottom-up approach will be used, not because it is better, but because it allows the use of examples that are confined to the point under discussion, without the distraction of a large "target language" to be learned before examples may be made.

The smallest things a compiler must reasonably be expected to deal with as it's
groups of characters taken together are usually the smallest things that have individual meaning in a language. For example, almost all programming languages use indentifiers, or variable names, made up by the user. The "rules" for these identifiers might be expressed in english:

> A letter, followed by none or more letters or digits, ended
> by the first character that is not a letter or a digit, is
> an identifier.

In META you could describe this with:

```
IDENTIFIER = .ACHR $ .ANCHR .QTOKEN ;
```

Which translates back into english as:

IDENTIFIER =	an identifier is
.ACHR	a letter
$	followed by none or more
.ANCHR	letters or numbers
.QTOKEN	(make it a single thing from now on)
;	(Thats all, Folks!)

The process of making a compiler with META begins with describing the language in such pieces as these. The fundamental terms that start with a "." indicate assembly code "run-time" subroutines, several of which are provided with Meta for use by compilers that it generates. You may also add your own run-time subroutines that are used in exactly the same way.

3. The Syntax of a META Program

META is a recursively defined language. Each part of it) is defined using smaller pieces. When we get to the small pieces, we find that many of them are defined by using the "higher level" pieces. It is like a cat chaising its tail! Because of this, it is necessary to have an overall picture of META as a language BEFORE the language may be adequately explained. To do this, we will make "two passes" at the problem. The first description of META is a simplified example, and is intended to give an overall picture, but not a good definition of each piece. When that has been done, a more detailed definition of META will follow.

3.1. Productions

The fundamental structure in META Language is the PRODUCTION. A production is to META what a statement is to another language. A production defines the syntax for a single "piece" of your overall syntax, in terms of even more fundamental pieces. A simplified syntax description of a production is:

```
PRODUCTION = <identifier>  '= <choices> '; ;
```

This breaks down as follows:

PRODUCTION =	The syntax known as <production> is defined as being
'=	an equal sign followed by
<choices>	the syntax called choices
';	followed by a semicolon
;	(end of the definition)

One point of interest is that the META compiler is written in META. The above META production is itself written in META. See if you understand how the line:

```
PRODUCTION = <identifier>  '= <choices> '; ;
```

fits its own definition of a production!

3.2. Choices

The <choices> syntax specifies that one and only one of a list of syntax descriptions must be used. A simple definition of <choices> is:

```
CHOICES = <termlist> $ ( '! <termlist> ) ;
```

Which introduces two new terms. The braces () indicate that everyting inside them is to be considered a single term. The $ indicates that the next single term is to be repeated as many times as it is matched.

CHOICES =	The syntax called CHOICES is defined as
<termlist>	The syntax <termlist>
$	followed by none or more
(	of the following group:
'!	The character !
<termlist>	The syntax <termlist>
)	(end of the group to be repeated)
;	(end of the definition of CHOICES)

3.3. Termlists

A definition of <termlist> is:

```
TERMLIST = ( <test> | <action> ) $ ( <test> | <action> ) ;
```

TERMLIST =	The syntax called TERMLIST is
(<test> | <action>)	Either the syntax of <test> or if not that, then the syntax of <action>.
$	followed by none or more
(<test> | <action>)	choice of the syntax of <test> or <action>
;	End of the defintion of <termlist>. A <termlist> ends when the input does not fit the syntax of either <test> or <action>

If the first term in a termlist fails, then control is returned to the choices level of syntax for testing the next choice, if any. However, if any term except the first term fails, then a SYNTAX ERROR is detected, and an error message will be generated. This is because each termlist is designed to handle a particular "phrase" and if part of it doesn't match, then there is an error. This may be overridden by placing the character ":" before any term, forcing a failure return as if that term were the first term. As an example, a numeric literal might be defined by:

```
NLIT = $ .blank  : .nchr $ .nchr ;
```

which states that any leading blanks are to be skipped, and then if the character is not a numeric digit, the term is not a numeric literal. If it is a numeric digit, then pick up any following digits also.

3.4. Tests and Actions

The syntax elements called <test> and <action> are the two fundamental terms of META. An action does something, such as generate output code, setting internal flags, etc. A test is a conditional action. It may either pass or fail. If a test passes, any characters that is used from the source code file are removed from the input stream. If a test fails, the source code input stream is unchanged from when the test started, with one exception. Many tests will skip over any blanks before starting, and these blanks ARE removed, even if the test fails. Later in this manual, individual terms are described, and those terms that do this are identified.

Some example tests are:

`TEST1 = '' <chr> ;`	Test for the existence of a single character. We used this above with '= to test for an equal sign
`TEST = '" <sl> '" ;`	Test for the existence of a string of characters, such as a keyword. "READ" would test for the keyword READ being next in the input stream.

Some examples of actions are:

`CODE = '! <sl> ;`	Generate output code from the pattern given in the string literal. An example: !"\bl\subroutine/".
`TEXT = ".TEXT" <sl> ;`	Send the string literal to the console as a message .TEXT "PLO Compiler V1.0".

These "mini-definitions" are intended to give you a frame of reference for the more detailed and accurate descriptions that follow. You should not expect to understand exactly how they fit together at this point.

4. Meta TEST terms

4.1. Single Character Test

SCTEST = '' chr ;

Any leading blanks are skipped. If the next character is the specified character, then the test passes, and that character is removed from the input stream. If it is not the specified character, then the test fails, and only the leading blanks have been removed from the input stream.

4.2. Multiple Character Test

MCTEST = <string literal>

A string literal specifies a multiple character test. Any leading blanks are skipped, and then the literal is tested against the input stream. If it matches, the characters are removed from the input stream, and the test passes. If not, only the leading blanks are removed from the input stream, and the test fails. If upper case conversion is enabled, the test literal MUST be specified in upper case to match the input stream.

4.3. Multiple Character Test with Delimiter Check

MCTESTD = '? <string literal> This test is identical to MCTEST except that the character that follows the last character of the matched string literal must NOT be alphanumeric if the test is to pass. This lets you test for a word such as GET and fail when scanning GETTING.

4.4. BLANK test

Since many META tests, including all of the above listed tests, skip any leading blanks that are present, while others, such as those used to build tokens, do not, the following test will pass if a blank is the next character, and if so, the blank will be removed from the input stream.

.BLANK

This is an example of an assembly language test reference.

4.5. Assembly Language Tests

Any term that starts with a period and is followed by an identifier is considered a call to
called with a BL instruction and returns with the EQ flag set to indicate FAIL, and with the EQ flag cleared to indicate PASS. Registers r6 and r7 are used for scanning characters and must not be changed, and register r10 is a local use stack that may be used but must be restored upon return. See the source code for the METALIB routines for examples.

```
  ASMTEST = '. <identifier>  [ <args> ]
```

The optional arguments are defined by:

```
  ARG = <numeric literal> ! <identifier> ! <string literal>
        ! '' .anyc  ;
```

And represent parameters passed to the routine by generating them as inline data statements following the BL instruction.

As an example, the test .ASMEXAMPL(1234,alpha,'c) will generate the following call:

```
          bl        ASMEXAMPL
          data      1234
          data      alpha
          data      "c"
```

And the term .ASMSTG("string of text") will generate:

```
          bl        ASMSTG
          text      'string of text'
          byte      0
          even
```

4.6. Invert Pass/Fail

If any test term is preceeded by a minus sign, then it's pass/fail status is reversed. For example, -'" means to test for a quote character, and remove it if present. Fail if it was present, and pass otherwise.

4.7. Discard Tokens

```
DTOK = '^ '( <numeric literal> ') ;
```

The indicated number of tokens are removed from the token stack and discarded.

4.8. Production Call

An identifier that does not have a period before it is a call to another production. This lets you de in pieces and connect them. The pass/fail status of that production becomes the pass/fail status of the term. An example of this is the use of <arg> in the specification of an assembly language test. Note that the characters < and > are optional, as they are allowed for compatibility with BNF notation only. Usually, they are not used.

4.9. Nested levels of CHOICES

Anyplace that you may use an individual test, you may use a set of choices, by enclosing them in {braces}.

4.10. Syntax of TESTS

```
TESTS = <identifier>   % production call %        [redacted]
        ! <string literal>  % multi-character test %
        ! '? <string literal> % test with delimiter check %
        ! '- tests          % invert pass/fail of next term %
        ! '^ '( <integer literal> ') % discard tokens %
        ! '. <identifier> [ arg ]  % assembly language test %
        ! '' chr            % single character test %
        ! '{ choices '}     % outer level choices as a term %
        ;
```

5. META ACTION Terms

ACTION Terms are those terms that always pass, and thus are not tested. They perform some desired action. They are used to generate output code, make messages, provide optional constructs, and repeat parts of the syntax.

5.1. Counted Repeat

This term provides the ability to repeat a selected term and count down the value stored in a .DECLARE variable. When the value is zero, the repeating ends. The format is:

```
  RPT = ?"REPEAT" <declare cell identifier>
        ( action ¦ test ) ;
```

5.2. Message Generating Terms

```
  .ERROR <string literal>
  .TEXT <string literal>
```

Both of these terms display the string literal as a console and listing message. Error will also generate a syntax error sequence.

5.3. Optional CHOICES

By enclosing a term or a list of choices seperated by "¦" in [brackets], the resluting pass/fail status is ignored, making it's presence optional. Note that this does not mean that a multiple term choice that passes it's first term can fail following terms.

5.4. Repeat Term until Fail

```
 RF = $ <term>
```

The term is repeated until it fails, and the fail status is converted to pass.

5.5. CALL Trace Control

```
  .TRACE
  .NOTRACE
```

These terms turn a trace listing of each production as it is called, on and off. This is used to debug your META program. These terms should not be in any finished META program.

6. Output Code Generation

As the syntax analysis of the source code progresses, appropriate assembly language code should be generated to perform the statements. Code may be sent directly to the output stream (usually the TEMP1$ file) or it may be stored in memory (deferred) for later output. This is useful when the source syntax is in a different order than the code that must be generated. An example of this is a statement to write data to a disk file:

```
PRINT #1;A,B,C
```

The code to write a line to the disk file will be generated by analyzing "PRINT #1;" but should not appear in the assembly program until after the line to be printed has been edited by analyzing "A,B,C". In this case, the output from the "PRINT #1;" is deferred until after the output from "A,B,C" has been generated.

META version 3.2 offers 4 seperate deferred output streams, and also offers a switchable output stream. The switchable stream may be assigned to direct output or to any of the deferred output streams, and then other productions that generate code to the switched output stream will use the pre-selected output stream. An expression analyzer might generate code to the switched stream. Other productions then could reference general expressions and select which output stream the expression code would be sent to.

When you are ready to use the code that has been sent to a deferred output stream, you transfer all code saved in that stream to the direct output stream. In the above example, the sequence of events might be:

Generate code for "PRINT #1;" to a deferred output stream
Generate code for "A,B,C" to the direct output stream
Transfer all code in the deferred stream to the direct stream.

Transfering a deferred output stream empties it. It may then be used again for new deferred output code.

6.1. Code Generation ACTION terms

The form of the direct output ACTION term is:

```
DCODE = '! <string code literal>
```

The form of a deferred output ACTION term is:

```
DEFCODE = '! <numeric literal> <string code literal>
```

For the present version, the numeric literal must be 1,2,3, or 4.

To transfer code from a deferred output stream, use:

```
DEFTRAN = '^ <numeric literal>
```

The numeric literal must be 1,2,3, or 4.

The form used to select the switched output stream is:

```
SWSEL = '!  '= <numeric literal>
```

The numeric literal must be either 0 for direct output, or 1,2,3, or 4 for deferred output.

To generate code to the switched output stream, use:

```
SWCODE = '! '0 <string code literal>
```

6.2. String Code Literals

The actual code to be generated is specified by a string& code literal. This is a text string enclosed in "quotes". Several characters have special meanings in such a string.

\	Tab to next assembly field
/	end the line of assembly code and send it to the output stream
´c	copy the next character exactly. This is used to output characters that have other meanings.
*	output the top token and remove it from the token stack.
+	output the top token, but leave it on the token stack.
#0	Generate a decimal number for the value in OUTO.
#n	Generate a label unique for this production call. There are four such labels available for each production iteration.

All other characters are copied exactly as they appear.

For each of the following examples, assume that NAME is on the top of the token stack, and ADRS is next on the token stack.

-
  ```
  !"\pshr\r0/"
  pshr        r0
  ```

-
  ```
  !"\li\r0,*/\mov\r0,*/"
  li          r0,NAME
  mov         r0,ADRS
  ```

-
  ```
  !"\li\r0,´"´*´"/"
  li          "*"
  ```

-
  ```
  !"\mov\+,r0/\mov\´*r3´+,*/"
  mov         NAME,r0
  mov         *r3+,NAME
  ```

7. OPTIONS and SETUP statements

There are several meta facilities that require setup or data declaration before starting your program. Collectively, these are called options, even though some of them are very necessary. They appear in your META program before the .SYNTAX or .STATEMENTS terms.

7.1. FILEID

One such setup option is the assignment of a file id for use by the link editor. Each META program module should start with this option:

```
   .FILEID <module identifier> ;
```

7.2. FILETYPES

Another setup option that must be present in a main module only (one that has .SYNTAX in it) is the filetype option. This specifies the default file types to be used for source and destination files if the names given do not have periods in them. It's format is:

```
  .FILETYPES  .<source file type> . <reloc file type>
              <exit cmd name> ;
```

As an example:

```
  .FILETYPES .MET .REL ASM ;
```

is used by the META compiler itself.

Use of an exit command name other than ASM allows code optimizer modules to be automatically included in the compilation process.

7.3. Attributes

There are two types of attributes. GLOBAL attributes are general purpose yes/no flags. SYMBOL attributes are yes/no flags that are related to an individual identifier. There are 32 global attributes and, for each identifier, there are 32 symbol attributes.

To declare an attribute, use the .attribute statement:

```
  .attributes name lit [, name lit ... ] ;
```

where name is an identifier associated with the attribute, and lit is the numeric bit number 1 through 32 assigned to that attribute. Some examples:

```
  .attributes fpvar 1, intvar 2, stgvar 3;
  .attributes inpfile 25, outfile 26;
```

Each attribute becomes an assembler equ statement:

```
  .attributes fpvar 1, intvar 2, stgvar 3;
```

translates into:

```
  fpvar  equ  1
  intvar equ  2
  stgvar equ  3
```

To use global attributes, you use the following terms:

.s(attribute)	set global attribute on
.r(attribute)	reset global attribute off
.if(attribute)	pass if global attribute is set (on)
-.if(attribute)	pass if global attribute is reset (off)

To use symbol attributes, you use the following terms, keeping in mind that they apply to the symbol that is closest to the top of the token stack:

.as(attribute)	set symbol attribute on
.ar(attribute)	reset symbol attribute off
.aif(attribute)	pass if symbol attribute is set (on)
-.aif(attribute)	pass if symbol attribute is reset (off)

Attributes (both symbol and global) are all reset upon loading your compiler, and if necessary, must be set by you.

7.4. Compiler Variables

You can set aside named integer variables for your compiler to use while compiling a program. You do this with the declare statement:

```
.declare name [(length)] [,name[(length)...] ;
```

where name is the name to be used by the variable, and should be unique in its first 6 letters, and length is the number of 16-bit words set aside for that name. If the length is not specified, then 1 word is set aside. Some examples are:

```
.declare nrint,nrfp;
.declare big(1000);
```

Each name is defined as an entry name so that the link editor may allow many modules to refer to that variable.

To use these compiler variables, the following terms are available:

.clr(var)	set var to 0
.inc(var)	add 1 to var
.dec(var)	decrement var
.set(var,lit)	set var=lit (the literal value)
.mov(var1,var2)	set var2=contents of var1
.max(var1,var2)	set var2= largest of var1 or var2

.eql(varlit1,varlit2)	pass if varlit1=varlit2

EQL treats each parameter as a literal if its value is 255 or less. Otherwise, it is assumed to be the address of a compiler variable, and the contents of that variable is tested.

```
.gend(var)
```

GEND generates a decimal number equal to the valu[illegible] var into the output stream.

Any externally defined variables in the compiler runtime package (metalib/metautil) may be manipulated with these terms.

There are three terms available for performing arithmetic on declare cells:

.cadd(var,lit)	add the literal to the variable
.vadd(svar,dvar)	add the source variable to the destination variable
.vmpy(svar,dvar)	multiply the two variables and store the result in the destination.

7.5. Utility Stacks

META 3 provides you with the ability to have several utility stacks under your direct control. To declare each stack use the statement:

```
.stacks name(length) [,name(length)...] ;
```

which declares each name a utility stack holding length number of 16-bit words.

To use these stacks, you have the folowing terms:

.spush(var,stack)	push var to stack. pass unless stack overflows.
.spop(stack,var)	pop var from stack. pass unless stack is empty (underflow)

7.6. Keywords

In most languages, there are certain keywords that must not be used for identifiers, as they are used by the language itself. The term .KWCHK described under tokens checks a list of such keywords. For this to work, however, the keyword list must be defined. The keyword statement does this:

```
KW = ?".KEYWORDS" <kwrd> $ <kywrd> '; ;

kwrd = .achr $ .anchr ;
```

All keywords MUST be listed in upper case to allow case insensitivity in the resulting compiler.

An example is:

```
.KEYWORDS GET PUT READ WRITE DO FOR TO STEP ;
```

7.7. Symbol Value Cells

Each symbol table entry may have one or more named value cells attached to it, which are all set to zero when the symbol is defined. You implement this with the .values statement:

```
   .values name [,name...] ;
```

There may be only one values statement per program, which must list all of the desired value cells.

For example:

```
   .values nrdim, tcode, assoc, syequ;
```

would declare that each symbol table entry will have 4 value cells, known as nrdim, tcode, assoc, and syequ, which might perhaps refer to the number of dimensions, variable type code, any associated variables, and some symbol equate value.

You may only work with the symbol value cells for the symbol that is closest to the top of the token stack. You do it with the following terms:

```
   .vld(var,valcell)  move variable to symbol value cell
   .vst(valcell,var)  move symbol value cell to variable
```

for example:

```
   .vld(intbin,nrdim)  move intbin variable to the nrdim cell
                       of the current symbol.
```

8. Source Stream Scanner Control

Several external variables are available in the input file scan routine to allow META programs to control the input stream. They may be changed with .SET and tested with .EQL.

eolchr	This cell holds the chaaaracter to be apended at the end of every source line. Set it to a space unless you have a line oriented language.
cmtchr	This character starts a comment. The input source stream is ignored until an end-fo comment character appears.
cmtend	This character ends a comment. If comments are handled by a statement type such as REM in BASIC, set cmtchr and cmtend to 0 to disable comments.
lflchr	This character appearing in the source stream will flush to the end of the line and set the next source line as if it were on the same physical line of text.
lflush	This switch causes the line flush action. If your program decides to ignore the rest of an input line, set this variable to 1.
symuc	If this switch is not zero, all characters except those accessed through .ANYC will be converted to upper case.
smode	This switch controls string mode. When it is non-zero, comments controlled with cmtchr and cmtend are temporarily disabled, so that those characters may be used in strings.
colcnt	This cell holds the column number of the character last accessed, starting with 1. If it is zero, the next character will be the first character on a line.

In addition there is one test term provided:

.NEOL

which passes if there are any characters left on the present line of source text.

9. Using the META Compiler

META (and all compilers written with it) have the following command syntax:

```
META <reloc file>=<sourcefile> [[,<asm file>] [,<listing file>]]
```

Relocatable files will have .REL appended to their name unless a period appears in the specified name. Source files will have .MET appended to their names unless a period appears in the name. (These default file types are determined by the .FILETYPES statement).

To use a file without any type default, specify the name with a period as the last character:

```
  META   temp2$.=program
```

If a compile only operation is desired, omit the relocatable file name:

```
  META   =program
```

There are a few "typing saver" options allowed with the relocatable and source file name. If no equal sign is present, then the first file name specified is used for both files:

```
  META program
```

will use program.rel and program.met

If the files are on different drives, you may use the form:

```
  META 1/=2/program
```

which will use 1/program.rel and 2/program.met

META 3.5 QUICK-REFERENCE SUMMARY

STRUCTURES

```
<prog>      =    [<options> ...]
                 (.STATEMENTS ! .SYNTAX )
                 $ <stmt> .END

<stmt>=    <id> '=   [ '! <termlist> ] <choices>

<choices> =      <termlist> $ ( '! <termlist> )

<termlist>=      <term> $ ( <action> ! ':<test> !<test> )

<term>      =    <action> ! <test>
```

OPTIONS

```
.FILETYPES .source .reloc exec
.TABS
.NOTABS
.STACKS <id> [ <id2> ] ( <n> ) ,...
.DECLARE <id> [ (<n>) ] ,...
.ATTRIBUTES <id> <n> ,...
.FILEID <id>
.CODE <id> <s> ...

.VALUES <id> ,...
.KEYWORDS <kid> [,] ...
```

ACTION TERMS (NOTEST)

```
!= <n>     assign variable output stream
           0 is direct output, 1-4 defered
!0 <s>     variable output from literal string
!0 <p>     variable output from code pattern
!<n> <s>   output to defered stream from literal string
!<n> <p>   output to defered stream from code pattern
^<n>       pop defered output stream <n>
.PRNDEF(<n>)    print defered stream on console as message

.REPEAT <v> <term>  perform <term> <v> times
.TRACE              production call trace on
.NOTRACE            production call trace off
.ERROR <s>          syntax error with displayed text message
.TEXT <s>           display text message
.FAIL               fail current production
.PASS               term that always passes
[ <choices> ]       optional choices
$ <term>            repeat term as long as it passes
.LIMIT <n> $ <term> repeat passing terms up to <n> times
```

TEST TERMS (can pass or fail)

<id>	invoke production
<s>	pass if string literal value is next instream
?<s>	as above, but delimiter must be non-an to pass
-<term>	invert pass/fail of <term>
^(<n>)	discard <n> tokens
^	discard one token
. <id>	invoke assembly language subroutine
. <id> (<args>)	asm subroutine with arguments
`<c>	test for occurance of character <c> next instream
'<c>	test for character, allowing leading blanks
(<choices>)	allow multiple choices as a single term

TOKEN BUILDING TERMS

.achr	alpha character builds
.anchr	alpha or digit ok
.nchr	digit ok
.hchr	hex digit ok
.anyc	any character ok
.untokn	remove char last appended to build buffer

pvchr	= character accepted by test
pvnum	= 0 thru 9 value of last chr if digit and 10 thru 35 for A thru Z

.mtoken('c)	if next chr is "c" then append it
.itoken('c)	append the character "c"

.kwchk	pass if token not a keyword if it is, return token to instream & fail
.qtoken	queue token to token stack

.fsymbl	pass if token is previously defined set CURSYM
.asymbl	add (define) token as new symbol set CURSYM
.rsymbl	reference symbol from CURSYM for attribute values, etc.

.symscn	initialize symbol table scan
.nxtsym	append next symbol to build buffer normally followed by .qtoken sets CURSYM

CURSYM	current symbol pointer

CHARACTER CLASS VARIABLES

The character classes are:

[illegible]	CCUCA	Upper Case Alpha
2	CCLCA	Lower Case Alpha
4	CCN	Numeric Digit
8	CCH	Hex letter A-F or a-F
16	CSPCL	Special Characters
3	CCA	Alpha upper or lower case
7	CCAN	Alpha or numeric digit
12	CCHN	hex digit 0-9, A-F, or a-f
32		(unused)
64		(unused)
128		(unused)

CHRACTER CLASS OPERATIONS

.CLTEST(<v>,<classvar>)	pass if char in v fits class
.CLCOPY(<oldclass>,<newclass>)	Define <newclass> to be all characters fitting <oldclass>
.CLINS(<char or var>,<class>)	Add character to class
[illegible]CLDEL(<char or var>,<class>)	Remove character from class

ATTRIBUTES

.s (<id>)	set global attribute
.r (<id>)	clear global attribute
.if (<id>)	test global attribute
.as (<id>)	set symbol attribute
.ar (<id>)	reset symbol attribute
.aif (<id>)	test symbol attribute

VARIABLES (declared)

.clr(<v>)	clear variable to 0
.inc(<v>)	add 1 to varaible
.dec(<v>)	subtract 1 from variable
.set(<var>,<n>)	set variable to value <n>
.mov(<fromv>,<tov>)	tov=fromv
.max(<v1>,<v2>)	v2=max of the two variables
.eql(<v1>,<v2>)	pass if v1=v2 values less than 256 are literals otherwise they are variable addresses
.gend(<v>)	output decimal value of <v> direct
.cadd(<v>,<n>)	add literal <n> to variable <v>
.vadd(<v1>,<v2>)	add <v1> to <v2>
.vmpy(<v1>,<v2>)	v1*v2 to v2
.vlt0(<v>)	pass if v<0
.evenup(<v>)	round V up to next even value
.dadd(<v16>,<v32>)	add 16 bit v16 to 32 bit v32
.dmpy(<v16>,<v32>)	multiply 16 bit v16 to 32 bit v32
.dneg(d32)	negate 32-bit variable

STACKS

.spush(var,stk)	push integer to stack fail if stack is full
.spop(stk,var)	pop stack to integer fail if stack is empty

VALUES of symbols

.vld(var,valuename)	set symbol value
.vst(valuename,var)	set symbol value to var

SCAN CONTROL

.NEOL	pass if not end of line
.BLANK	pass if next character is a blank
.UNSCAN	unscan previous character chr must be on same source line

eolchr	chr to append at eol
cmtchr	chr to start embedded comment
cmtend	chr to end embedded comment
lflchr	char to flush rest of line
lflush	switch to flush line if not 0
symuc	convert to uppercase if not 0, except .ANYC
smode	string mode - disables cmtchr, cmtend
colcnt	col # of last chr accessed. 0=start of line next

OTHER STANDARD VARIABLES

nolink	# errors. If 0, compiler will link to next program
nos$	0=mdex -1=NOS
out0	used to hold value generated in output

CODE GENERATION ELEMENTS

\	tab to next ASM field
/	end generated line
*	use token from stack
+	copy token from stack
'c	use c literally (used to output CGEN characters)
#0	generate OUT0 value in decimal
##	generate OUT0 value in hexidecimal
#1	generate label unique to production
#2	
#3	
#4	

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 [illegible] 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 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 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

```
        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

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
                         time through.
A91
        LD    I
        FLP      ]       Test for first time through.
        BFP   A92]
        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

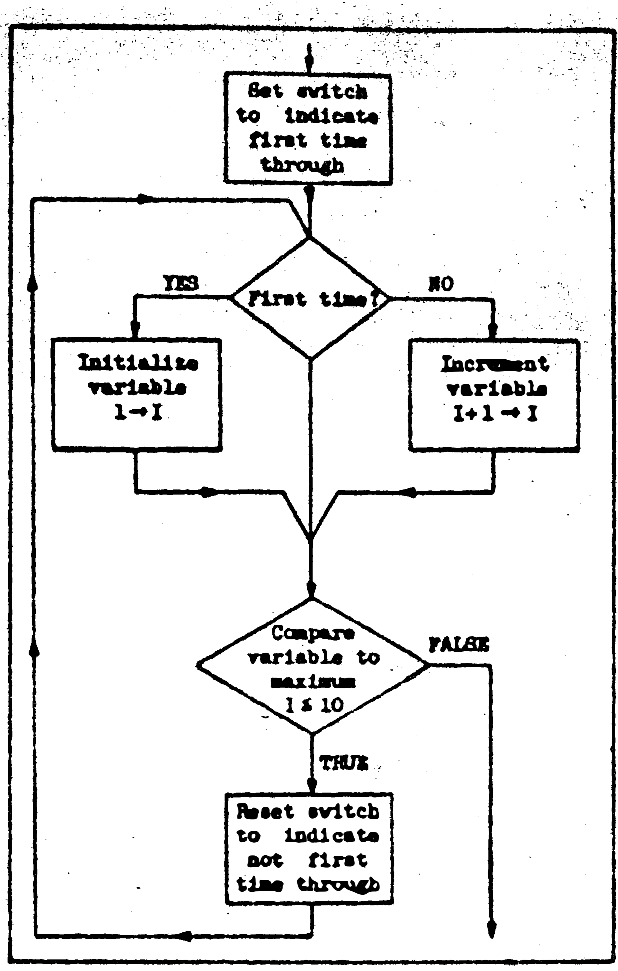

Illustration 4

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 ororder 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 5

```
.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 6

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 7

```
.BEGIN
.REAL X ., 0 = X .,
                B    A01                                      0000 G 0012
        X                                                     0004
                BLK  001                                      0004
        A01                                                   0012
                LDL  0                                        0012 A
                ST   X                                        0021 D 0004
.UNTIL X .= 3 .DO .BEGIN
        A02                                                   0025
                LD   X                                        0025 B 0004
                LDL  3                                        0029 A
                EQU                                           0038 F
                BTP  A03                                      0039 K 0097
      EDIT( X*X * 10 + 1, '*') ., PRINT ., X = X + 0.1
                LD   X                                        0043 B 0004
                LD   X                                        0047 B 0004
                MLT                                           0051 E
                LDL  10                                       0052 A
                MLT                                           0061 E
                LDL  1                                        0062 A
                ADD                                           0071 C
                EDT  01'*'                                    0072 I
                PNT                                           0074 G
                LD   X                                        0075 B 0004
                LDL  0.1                                      0079 A
                ADD                                           0088 C
                ST   X                                        0089 D 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 7
FIGURE 8

D.V. SCHORRE - ACM PROCEEDINGS 1964

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 COUNTER TO CARD COLUMN 8.

Figure 6.2

CONSTANT AND CONTROL CODES

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

D.V. Schorre - ACM-Proceedings 1964

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

```
                                   IMPLICATION   IMPLICATION
Q   BEQUIV = IMPLICATION $('.EQ' .OUT('EQU')) ..

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

    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') ';'
         ST .OUT('R  ' *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.

omit

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.
PRT	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 CHARACT- [illegible]
HLT	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 5

```
.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(Z) .,
     ABS = .IF 0 .L= Z .THEN Z .ELSE -Z .,
.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 N, V, T .,  .ARRAY MATRIX (. 0 .. 20 .) .,
.UNTIL .FALSE .DO .BEGIN
     EDIT(1, 'FIND DETERMINANT OF' ) ., PRINT., PRINT.,
     READ(N, 1) .,
     .FOR V = 0 .STEP 1 .UNTIL N-1 .DO .BEGIN
          READ(MATRIX (. N*V .), N) .,
          WRITE(MATRIX (. N*V .), N) .END .,
     PRINT .,  T = DETERMINANT (MATRIX, N) .,
     WRITE(T, 1) ., PRINT., PRINT .END
.END PROGRAM
```

META-3

A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code

by Frederick W. Schneider and Glen D. Johnson, UCLA Computing Facility, Los Angeles

ABSTRACT

The basic compilation method is a top to bottom recursive scan without backtrack based on the compiler written for the IBM 1401 by Val Schorre. Each statement of the language is written in a form closely resembling Backus Normal Form; that is, a sequence of tests to be performed to determine whether or not the sequence of characters in the input string is a valid program in the language described. In addition, output instructions are interspersed with the syntactic elements to generate the desired code. The following features were added to the language to facilitate the direct generation of efficient machine code:

1. *A symbol table*
2. *A push-down operand stack*
3. *Mode flags and a register manipulation generator*
4. *A push-down first-in first-out list*
5. *Direct communication in a simplified manner between the compiler and hand coded routines.*

A complete description of both the META-3 compiler and of the compilation algorithm are given.

META-3

Contrary to popular opinion, syntax-directed compilers can rapidly generate quite efficient machine code for machines without push-down hardware. The method used in our compiler [illegible]sed on the META II compiler developed by D. V. Schorre [illegible]e IBM 1401, but it is modified to facilitate direct generation of sequential code rather than polish-like code.

The META-3 compiler constructs a series of tests and references to external routines from an input language resembling Backus Normal Form, with code defining clauses added. This construction assembles into a compiler for the language defined.

Two types of operations are basic in the meta-language: actions and tests. An action is an unconditional operation such as outputting, setting flags and so forth. There are two major types of test. One is to test internal status such as the type of a variable, the other is to test the input stream for the occurance of an identifier, a specific character string, or a general form of string. Each test returns the value *true* or *false* depending upon whether the tested condition was met or not. The Meta-compiler generates the code to test this value after every test and either proceeds, if true, or, if false, trys to return the value *false* to the caller. Since anything tested for and found is deleted from the input stream, any *false* return other than the first of a sequence of tests will be made to transfer control to a diagnostic routine which prints the top element of the stack, the present status of the input stream, and a complaint about bad syntax. The discussion of the syntax equations for META-3 as written in META-3 will show the usage and definition of the basic syntax elements. For a further discussion of the basic algorithm or references on the subject see Schorre's paper in this volume.

[illegible]ach syntax equation begins by naming the construct which [illegible]s defining, and ends with a semicolon (written '.,'). The definition is a series of tests and actions, which may be grouped by parenthesization. A string in quotes (e.g. 'STRING') is a test which is *true* only if the specified characters appear next in the input stream. '.ID' is a test which is *true* only if an identifier is the next thing in the input stream. An identifier is an alphabetic character followed by a series of alphabetic or numeric characters, and terminated by the first unrecognizable character, usually a blank. The first six characters of the identifier must be unique and are the only portion of the identifier retained. '.ID' causes this identifier to be placed in the push-down operand stack. Alternate definitions are identified by separating them with slashes (/). For simplicity of writing the sequence operator '$' is used to reduce the number of recursive definitions needed, and is read: ' a sequence *(which may be empty)* of...'. The test following the sequence operator is performed until it returns *false*, at which point the sequence is satisfied. 'EMPTY' is a test which always has the value *true*. An identifier indicates a syntactical structure, usually defined by another equation, which is to be tested for. '.STRING' is a test which removes a string from the input stream, assigns it storage, and places its symbolic address (of the form .Z.nnn) in the operand stack.

Outputting is indicated by the '.OUT' or '.CALL' verbs. '.OUT' is followed by a list, in parentheses, of output arguments to be placed in the fields of a symbolic card to be turned over to the assembler. There are three fields; label, operation, and variable. Fields are separated by commas, and cards are terminated by slashes. There are three forms which each argument may take:

a) strings to be inserted litterally
b) '*' indicating the uppermost element of the stack
c) '*n' indicating the n^{th} label stack, each of which has a unique constant value at each usage of each statement.

'.CALL(...)' is eqivalent to '.OUT(,'CALL',...)' and generates the op-code CALL with the first argument going in the variable field.

Since the compiler is fully defined by its syntax equations, the following discussion of each equation will complete the description of the META-3 compiler.

.SYNTAX PROGRAM

Defines the principal syntactic element of this compiler.

PROGRAM =

Begins the definition of the syntactic element 'PROGRAM'

'.SYNTAX'

Tests the input stream for the quoted string. If *false* (since this is the first test of this definition) 'PROGRAM' will be *false*.

.ID

Tests for an identifier in the input stream. If found the first six characters are placed in the operand stack, and the entire identifier is deleted from the input stream. If not found the diagnostic routine is entered.

.OUT(,'ENTRY',*)

Outputs a symbolic card with the op-code ENTRY and the variable field containing the identifier in the top of the stack. The stack is popped up.

$ ST

Tests for the syntactic element 'ST' (defined below, and keeps going back for more until they are exausted.

'.END'

Removes the string '.END' from the input stream, giving a diagnostic if not found.

.,

End of statement.

The entire statement discussed so far is:

```
PROGRAM ='.SYNTAX'  .IC .OUT(,'ENTRY',*)
                    $ ST '.END' .,
```

It may be expressed in Backus Normal Form as:

```
<program> ::= .SYNTAX <identifier> <stseq> .END
<stseq> ::= <st>   <st> <stseq> ! <empty>
```

The equations for META-3 continue:

```
ST = .ID .OUT( *,'PXA','.4/, 'CALL','..PUSH')
```

This is the beginning of the defination of a statement and says that a statement starts with an identifier which is output as the label of a PXA ,4 instruction, then followed by a call of ..PUSH.

```
    '=' EX1 '.,' .CALL ('..POPP') .,
```

The identifier must be followed by an equal symbol and an EX1 (see below) and terminated by a At this point a call of routine ..POPP is output. The routines ..PUSH and ..POPP handle the recursion.

```
EX1 = EX2 $( '/' .OUT(,'ZET','..TEST' /,'TRA', *1) EX2)
  .OUT (*1,'NULL') .,
```

An expression one is defined to be an expression two followed by a sequence (*which may be empty*) of slash, at which point output a test for the truth of the previous expression, which, if met, will cause transfer of control to the label contained in "*1" which will be defined later After the slash must come another expression two. When there no more alternatives in the stream, define the label in "*1" by outputting it on a NULL.

```
EX2 = ( TEST .OUT(, 'NZT' , '..TEST' /, 'TRA',*1)/ACTION
    ) $( TEST .OUT(,'NZT','..TEST'/,'CALL','..DIAG')/
   ACTION) .OUT( *1 ,'NULL') .,
```

An expression two consisists of a number of tests or actions. If the first of these is not met the restof them are skipped. If any of the others is not met ..DIAG receives control.

```
TEST = .ID .CALL( *)
```

A test is defined to be either an identifier, in which case a call to the identifier is output.

```
   / '.ID' .CALL('..IDNT')
```

or the string '.ID' is which case a call on routine '..IDNT' is generated

```
   / '(' EX1 ')'
```

or, a left parenthesis followed by an EX1 followed by a right parenthesis

```
   / .STRING .CALL('..CMPR(' * ')' )
```

or, a string in quotes whose location is inserted into the stack and then output as an argument to ..CMPR

```
   / 'S.TRING' .CALL ('..STRT')
```

or, the word .STRING which causes a call on ..STRT to be generated

```
   / .CLA'( '-' DIGIT ALPHABETIC *1 .CALL('..CLAD(-'
      ',-H' *1 '*****)' )
```

or, the word CLA followed by a minus sign and a digit and a letter both of which are placed in the stack as they are found by external routines with the entry points 'DIGIT' and 'ALPHAB' These two characters are output as arguments of a call on routine ..CLAD by placing the letter in the *1 label stack and referencing it in the .CALL statement

```
     / DIGIT ALPHABETIC *1 .CALL('..CLAD(-' * ',-H'
      *1 '*****)' ) )
```

or the .CLA could be followed by just the digit and letter without the minus sign and receive a approximately the same treatment, except that the digit is transmitted to ..CLAD as positive rather than negative.

```
  / '-' ALPHABETIC .CALL('..MINS(-H' * '*****)' )
```

Or, a test may be a minus sign followed by a letter. In this case a call on ..MINS with the letter as an argument is generated. This is used with the symbol table discussed below.

```
   / '*' DIGIT .CALL('..MOVE(' * ')' )
```

Or, an asterisk followed by a digit which is compiled as an argument to routine ..MOVE. This is the routine which moves identifiers between the operand stack and the label stacks

```
   / '-' DIGIT .CALL( '..MOVE(-' * ')' )
```

Or, the asterisk could be followed by a minus sign and a digit which is given as an argument to move the "n" stack to the operand stack.

```
  / ALPHABETIC .CALL ('..STAR(-H' * '*****)' ) )
```

Or, the asterisk could have been followed by a letter which is compiled as an argument to ..STAR. Again, this is for the symbol table q. v.

```
  / '.T' ALPHABETIC .CALL('..SETT(-H' * '*****)' )
```

Or, the final thing which a test may be is .T followed by some letter which is used as an argument in the generated call on .SETT. This test references the mode flag.

```
ACTION = OUTPUT
```

An action is defined to be either an output (defined later).

```
   /'.EMPTY .OUT( 'STL' , '..TEST')
```

or, .EMPTY in which case the test cell ..TEST will be set non-zero to indicate that indeed an empty has been found.

```
   / '$' .OUT(*2, 'NULL') TEST
```

Or, a dollar sign, at which point the label in *1 is output,followed by a test (defined above)

```
     .OUT(,'ZET','..TEST'/,'TRA',*1/,'STL','..TEST')
```

after which test, if it was met it will be repeated, otherwise, ..TEST is set non-zero to indicate *true*.

```
   /'.STO' ALPHABETIC .CALL('..STOR(-H' * '*****)' )
```

Or, .STO followed by a letter which is compiled as an argument to ..STOR.

```
   /'+' ALPHABETIC .CALL('..PLUS(-H' * '*****)' )
```

Or, a plus sign followed by a letter which compiles as a call on ..PLUS with the letter as an argument (set attribute register to indicate this property).

```
   /'.S' ALPHABETIC .CALL('..SETS(-H' * '*****)' ) .,
```

Or, finally, an action may be .S followed by a letter which becomes, at object time, an argument to ..SETS (which sets the mode flag for later testing with .T).

```
OUTPUT=
```

An output is defined to be

```
    '.OUT' '(' OUT2 ')'
```

.OUT followed by an OUT2 in parentheses

```
  /'.CALL' .CALL('..FELD').CALL('..LITG(-02321434377001')
       .CALL('..FELD')
```

Or, .CALL which generates thesame instructions as .OUT(,'CALL', ...)

```
      '(' $OUT1 ')' .CALL('..PUBG')
```

followed by a parenthesis and a sequence (*which may be empty*) of OUT1's after which a call on ..PUBG is generated

```
   /'.ERITE' '(' DIGIT $OUT1 ')' .CALL('.ERITE(-'*')'):,
```

Or, finally an output may be .ERITE followed by, in parentheses, a digit and a sequence (*which may be empty*) of OUT1's, in which case the digit is given to .ERITE as an argument.

```
OUT2 = OUT2A $ ('/' .CALL('..PUBG') OUT2A) .CALL
       ('..PUBG') .,
```

An OUT2 is defined to be an OUT2A followed by a sequence (*which may be empty*) of slash (at which point a call to ..PUBG is output) followed by OUT2A's, at the end ..PUBG is called.

```
OUT2A = $ OUT1 $(',' .CALL('..FELD') $OUT1) .,
```

An OUT2A is defined to be a sequence (*which may be empty*) of OUT1's followed by a sequence(*which may be empty*) of commas (output a call on ..FELD) followed by sequence (*which may be empty*) of OUT1's.

```
OUT1 = '*' (DIGIT .CALL('..GENR(="*")' )
```

An OUT1 is defined to be either an asterisk followed by a digit, in which case ..GENR is called with the digit as an argument,

```
        / .EMPTY .CALL('..COPG' ))
```

or else, for an asterisk alone, a call on ..COPG is output.

```
    / .STRING .CALL('..LITG(' * ')' ) .,
```

Or, finally, an OUT1 may be a string whose location is compiled into a call on ..LITG.

```
.END          ..Signals the end of compilation.
```

Direct Communication Between Hand Coded Routines and the Meta-compiler

While compiling the meta-language description of a compiler, any identifier is assumed to be the name of a meta-linguistic variable, and, as such, has a call to it generated. Upon return the cell ..TEST is tested for the *true* or *false* result of the test performed. The IBMAP assembler assumes that any undefined symbol will be defined as an entry point to some other deck at load time.

This rather rash assumption on the assemblers part allows operations to be added at will with the understanding that if the added routine is actually only an action the compiler still treats it as a test, and tests cell ..TEST on return from the routine, and had better find it non-zero (or *true*) at that point if stray error messages are to be avoided and compilation is to continue.

The Push-down Operand Stack

The meta-linguistic element * is to be treated as a push-down stack. Whenever an identifier is successfully discovered it is placed on the top of this stack. It may be removed (and the stack popped up) either by having* in an output imperative, by the FIFO, or by entering subroutine REMOVE which may be called either from a syntax equation or from a hand coded routine.

This stack is extended by allowing copies to be freely made from the * stack to any one of the four local safe cells (*1, *2, *3, *4) and also allowing back-copying (*-1, *-2, *-3, *-4)..

All other operands such as strings, digits, etc. are entered as the topmost element of the stack as they are discovered in the input stream.

The FIFO

An interesting technique implemented in META-3 is the combined push-down and first-in first-out list. The operations FEE, FI, FO, and FUM are used to address it, the elements being inserted by FI and removed by FO. FEE is used to push the list down and insert a level mark, while FUM generates a call statement on the variable in the top of the operand stack, with all the elements in the top of the FIFO as arguments.

The basic structure of the list is that of a number of superimposed FIFO lists(or queues).

FI removes the uppermost element of the operand stack and inserts it into the FIFO list as the last element.

FO removes the first element of the FIFO list and inserts it as the uppermost element of the operand stack, however if the present queue or FIFO area is empty FO will return *false* and pop the stack to the underlying FIFO.

FEE starts a new list, marking the top of the previous one. That is it pushes down the previos queue, and starts a new one on top of it.

FUM generates a call to the uppermost element of the operand stack and gives as arguments all the elements of the uppermost FIFO list(if any).

The following example of the use of this list will give an idea of its use. The first column represents the contents of the operand stack, the second the operation in the compiler, and the third the contents of the FIFO.

Operand stack	Operation	FIFO
A B Γ Δ E	FI	*empty* initially
A B Γ Δ	FI	E
A B Γ	FI	ΔE
A B	FEE	ΓΔE
A B	FI	ΓΔE|
A	FI	ΓΔE|B
empty	FO	ΓΔE|AB
B	FO	ΓΔE|A
B A	FO	ΓΔE|
B A	*(returns false)* FUM	ΓΔE
B	*(outputs* CALL A(Γ,Δ,E))	*empty*

Intricate rearrangements and rescanning are possible using this list since anything not wanted now can be FIed and when needed restored in the identical order using a FO.

As is evident in the discussion the principle use of this list in the present version is processing procedure declarations and references, since, for compatibility with the rest of the world it is desirable to have the arguments appear in the object code in the same order as in the source program. Variables in the declaration can be cycled through the FIFO in order to pass a number of tests by doing alternate FOs and FIs and rolling up without changing the length of the area (this action reminds me of a tracked vehicule), while determining the parameters necessary for storage allocation.

The Symbol Table

This routine stores and examines symbols given to it. Each symbol may have any of 26 arbitrary attributes, represented as A through Z. These properties are given to the routine by or'ing a property into the attribute register. For example +R adds the property R to those properties already in the attribute register. The meta-language primitive CLEAR sets all the properties to *false*. The meta-language primitive SET nondestructivly places the element at the top of the * stack into the symbol table along with the contents of the attribute register. The OR verb gives the * stack identifier the properties represented by the contents of the attribute register in addition to its other properties.

The symbol table may be tested with the - property. This returns *true* if and only if the input string is an identifier with the given property. For example -P would test the input string for the property P. mnemonically this could test it for being a procedure name in a statement such as:

X -RANDOM

where RANDOM is a previously declared procedure. The * stack may be similarly tested by saying for example: *P.

The symbol table is extended to cover block structured languages by marking it and skipping back to the last mark. The marking is done by the verb BEGIN; the popping by END. These may be nested until the symbol table overflows. In addition, Though there is no immediate use, for determining whether or not a variable is local to a block the verb LOCAL returns *true* if the last identifier tested was found in the symbol table before a mark was found.

The Register Manipulation Generator

The register handling routine generates register load, store, and exchange instructions and keeps track of the object time registers. The machine for which we are compiling is assumed to have six registers: an A register for addition, a Q register useful for division, an I register and a L register, both used for logical operations, an R register used for double precision work, and the N register which is a negative A register. It is assumed that that two registers cannot both contain information at the same time.

The safeguarding of the contents is caused by the imperative .STO*x* where *x* is a register name. This causes insertion of a dummy register in the * stack, and the maintenance of a pointer to this register in the * stack.

The loading of a register from * is accomplished by .CLA*nx* where *n* is the depth of the *stack that the storage reference is to be taken from and *x* is a register name. The previous register contents, if any, are preserved by the generation of a store instruction. Register exchanges are performed if necessary. The loading imperative is extended by allowing *n* to be preceeded by -. In this case the register exchange is performed only if it is a pure exchange; that is, the requested operand is already in the registers.

EXAMPLE:

```
.SYNTAX EXPR

EXPR = EXPR0 FREEAC .,

EXPR0 = EXPR1 $ ('+' EXPR1 (.CLA-1A/.CLA2A)
                .OUT(,'ADD',*) .STOA) .,

EXPR1 = PRIMARY $ ('*' PRIMARY(.CLA-1Q/.CLA2C)
                .OUT(,'MPY',*) .STOQ) .,

PRIMARY = '(' EXPR0 ')' / .ID .,     .END
```

Gives for either (A+B) *C or C*(A+B) the following code

```
        CLA  A
        ADD  B
        XCA
        MPY  C
        STQ  .T+000
```

And for the expression (A+(I+J)) gives:

```
        CLA  I
        ADD  J
        ADD  A
        STO  .T+000
```

The verb FREEAC causes the contents of the registers to be unconditionally emptied.

The verb TPUSH marks a stack used to retain the number of temporaries used in any block. At the end of the block the verb TPOP will generate the instruction:

```
    .T     BSS   n
```

where n is the number of temporaries used.

A more complex example of the use of many of the features of META-3 is the listing of the syntax equations for CODOL in the appendix. CODOL is a minimal compiler, designed more to have an assembly listing of less than ten pages than to be useful for computation, and has the severe drawback that in our haste to prepare it provision for constants was completely overlooked, but could be inserted by allowing REALNUMBER as a PRIMARY. This routine is a hand coded one designed for the ALGOL 60 compiler now under development using the successor to META-3, META-4.

Acknowledgements

We thank the UCLA Computing Facility for the generous use of their IBM 7094, and E. M. Manderfield, without whose flogging this paper would never have been written.

References

1. Schorre, Val, 'META II A Syntax-oriented Compiler Writing Language', 1964 ACM Natl. Conf.
2. Schorre, Val, 'A Syntax-directed Smalgol for the 1401' 1963 ACM Natl. Conf., Denver, Colo.
3. Irons, E. T., 'The Structure and Use of the Syntax-directed Compiler', Annual Revue in Automatic Programming, The Macmillan Co. New York.
4. Schmidt, L., 'Implementation of a Symbol Manipulator for Heuristic Translation', 1963 ACM Natl. Conf.
5. Bastian, Lewis, 'A Phrase-Structure Language Translator', AFCRL-Rept-62-549, Aug. 1962.

Appendix A

Instructions generated by register exchanges

Source	Destination A	N	Q	R	L	I	Storage
A	—	CHS	XCA	LDQ =0	XCA XCL	XCA XCL PAI	STO
N	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Q	XCA	XCA CHS	—	XCA LDQ =0	XCL	XCL PAI	STQ
R	—	CHS	XCA	—	XCA XCL	XCA XCL PAI	DST
L	XCL XCA	XCL XCA CHS	XCL	XCL XCA LDQ =0	—	PAI	SLW
I	PIA XCL XCA	PIA XCL XCA CHS	PIA XCL	PIA XCL XCA CHS LDQ =0	PIA	—	ST
Storage	CLA	CLS	LDQ	DLD	CAL	LDI	—

Note: The transfers between I or L and any of A,Q,R or N are for completeness only, there no convenient instructions for this.

Note: The imperative .STQN is illegal, there being no convenient instruction for this.

Apendix B

Subroutines used and their Functions

Name	Usage	Statement	Function
..POPP	-	ST	Saves location of caller for recursion
..PUSH	,	ST	Recursive return
..TEST	any test	EX1, EX2	Non-zero if *true*, zero if *false*.
..DIAG	any test	TEST	Prints stack, input stream and nasty message about bad syntax.
..IDNT	.ID	TEST	Test for an identifier in the input stream, places first six characters in the stack if found.
..COMP	'XYZ.....	TEST	Tests for a string in the input stream returns *true* on match.
..STRT	.STRING	TEST	Tests for any string in the input stream, outputs it as: USE .STRN. .Z.nnn BCI m, the string USE PREVIOUS and places .Z.nnn in the stack.
..CLAD	.CLAnx or .CLA-nx	TEST	Entry point to register manipulator for register loads and exchanges.
..MINS	-x	TEST	Used to test input string and compare it with the symbol table.
..MOVE	*x or *-x	TEST	Moves single elements from the stack to the label stacks and vice-versa.
..STAR	*Z	TEST	Used to compare the stack with the symbol table.
..SETT	.Tx	TEST	Used to test the mode flag.
..STOR	.STOx	ACTION	Tells the register manipulator to hang onto the contents of x.
.PLUS	+x	ACTION	Sets the symbol table.
SETS	.Sx	ACTION	Sets the mode flag to X.
..FELD	,	OUTPUT	Begins a new field on output.
..LITG	'..........'	OUTPUT	Moves a fixed string into the output stream.
..PUBG	.OUT or .CALL	OUTPUT	Ends a card image.
.BITE	.ERITEn	OUTPUT	Writes an error message.
.GENR	*n	OUT1	Moves the label from the *n label stack to the output stream.
..COPG	*	OUT1	Moves the stack to the output stream and pops it up.
DIGIT	n	TEST	Moves one character of the specified type into the stack. DIGIT and ALPHAB may return *false*, CHARAC never.
ALPHAB	x	TEST	
CHARAC	x or n	ACTION	
CLEAR		ACTION	Resets attribute register.
TPUSH		ACTION	Marks the beginning of a block of temporaries.
TPOP		ACTION	Ends a block of temporaries and allocates storage to them.
USE		ACTION	Begins block of separate code.
USEPOP		ACTION	Ends block of searate code and returns to previous block.
OR		ACTION	Defines a symbol in the stack with the properties in the attribute register, and puts it in the symbol table.
SET		ACTION	
REMOVE		ACTION	Deletes the top of the stack.
FEE		ACTION	Reference the FIFOlist (see text).
FI		ACTION	
FO		TEST	
FUM		ACTION	
REALN		TEST	Trys to get a double-precision floating-point number from the input stream.

Appendix C

7094 META-COMPILER COMPILED BY ITSELF.

```
.SYNTAX  PROGRAM
OUT1 =
     '*'   ( DIGIT  .CALL(  '..GENR(='  *  ')'  )
           /  .EMPTY  .CALL(  '..COPG'  )   )
/    .STRING  .CALL(  '..LITG('  *  ')'   )
.,
OUT2A =
     $  OUT1   $  (  ','  .CALL(  '..FELD'  )  $ OUT1   )
.,
OUT2 =
     OUT2A  $ (  '/'  .CALL(  '..PUBG'  )  OUT2A )  .CALL( '..PUBG'  )
.,
OUTPUT =
     '.OUT'  '('  OUT2  ')'
/    '.CALL'  .CALL(  '..FELD'  )   .CALL(  '..LITG(=0232143437700)'  )
               .CALL(  '..FELD'  )
          '('  $ OUT1  ')'   .CALL(  '..PUBG'  )
/    '.ERITE'   '('  DIGIT  $ OUT1  ')'  .CALL(  '.ERITE(='  *  ')'  )
.,
ACTION =
     OUTPUT
/    '.EMPTY'   .OUT( , 'STL' , '..TEST' )
/    '$'  .OUT(  *1 , 'NULL'  )      TEST
                .OUT( , 'ZET' , '..TEST'   / , 'TRA' , *1  /
                                             , 'STL' , '..TEST'    )
/    '.STO'  ALPHABETIC  .CALL(  '..STOR(=H'  *  '*****)'  )
/    '+'  ALPHABETIC   .CALL(  '..PLUS(=H'  *  '*****)'  )
/    '.S'  ALPHABETIC  .CALL(  '..SETS(=H'  *  '*****)'  )
.,
TEST =
     .ID   .CALL(  *  )
/    '.ID'   .CALL(  '..IDNT'  )
/    '('  EX1  ')'
/    .STRING  .CALL(  '..CMPR('  *  ')'   )
/    '.STRING'  .CALL(  '..STRT'  )
/    '.CLA'    (  '-'  DIGIT  ALPHABETIC  *1
                .CALL(  '..CLAD(=-'  *  ',=H'  *1  '*****)'  )
            /  DIGIT  ALPHABETIC  *1  .CALL(  '..CLAD(='  *  ',=H'
                *1  '*****)'  )       )
/    '-'  ALPHABETIC   .CALL(  '..MINS(=H'  *  '*****)'  )
/    '*'  ( DIGIT  .CALL(  '..MOVE(='  *  ')'  )
          / '-' DIGIT  .CALL(  '..MOVE(=-'  *  ')'  )
          / ALPHABETIC   .CALL(  '..STAR(=H'  *  '*****)'    )    )
/    '.T'  ALPHABETIC  .CALL(  '..SETT(=H'  *  '*****)'  )
.,
EX2 =
     (  TEST   .OUT( , 'NZT' , '..TEST'  / , 'TRA' , *1  )  /  ACTION )
$ ( TEST  .OUT( , 'NZT' , '..TEST'  / , 'CALL' , '..DIAG'  )
          / ACTION  )   .OUT( *1 , 'NULL'  )
.,
EX1 =
     EX2 $ (  '/'  .OUT( , 'ZET' , '..TEST'   / , 'TRA' , *1  )
               EX2  )    .OUT( *1 , 'NULL'  )
.,
ST =
     .ID   .OUT( * , 'PXA' , ',4'  / , 'CALL' , '..PUSH'  )
          '='    EX1   '.,'    .CALL(  '..POPP'  )
.,
PROGRAM =
     '.SYNTAX'  .ID   .OUT( , 'ENTRY' , *  )
               $ (  ','  .ID   .OUT( , 'ENTRY' , *  )
                 $ ST   '.END'  .,
.END
```

```
           CODOL COMMON DEMONSTRATION ORIENTED LANGUAGE
.SYNTAX PROGRAM

PROGRAM=.OUT('......','SAVE') TPUSH SEGMENT .OUT(,'RETURN','......')
         $(.ID *1 .OUT(*1,'SAVEN') SEGMENT .OUT('RETURN',*1))
                   TPOP .,

SEGMENT= DECLARATION'.,' $(DECLARATIONS '.,')  ST $('.,' ST ) .,

DECLARATION = 'REAL' .OUT(,'USE','.STOR.') CLEAR +R .ID SET
                        .OUT( *,'PZE') $(',' .ID SET .OUT(*,'PZE'))
                      .OUT(,'USE','PREVIOUS')
         /'FORMAT' .ID CLEAR +S SET *1 .STRING .OUT(*1,'EQU',*)  .,

ST = '*' .ID  .OUT( * ,'TRA','*+1') ST
   / 'GO' 'TO' .ID  .OUT ( , 'TRA' , * )
   / 'CALL' .ID FEE ( '(' EXPR FREEAC FI $( ',' EXPR FREEAC FI)  ')'
                    / .EMPTY )   FUM
   / 'SET' FEE .ID FI $( ',' .ID FI ) '=' EXPR .CLA1A
                      FO .OUT( ,'STO',*) $(FO .OUT(,'STO',* ) )
   / 'IF' EXPR .CLA1A ( 'PLUS'.OUT( ,'TMI',*1) /'MINUS' .OUT(,'TPL',*1)
                      / 'ZERO'.OUT( ,'TNZ',*1) /'NON''ZERO' .OUT(,'TZE'
                                          ,*1))   ST  .OUT(*1,'NULL')
   / 'ALTER' .ID 'TO' .ID .OUT(,'AXT',*',4'/,'SXA',*',4')
   / 'PRINT' .ID *S .CALL('.FWRD.(.UN06.,'*')')
                  $(',' EXPR .CLA1A .OUT(,'TSX','.FCNV.,4'))
   / 'READ' .ID *S .CALL('.FRDD.(.UN05.,'*')')
                    $( ',' .ID .OUT(,'TSX','.FCNV.,4'/,'STO',*)   )   .,

EXPR = '-' NEXPR / ('+'/.EMPTY) EXPR1  .,

EXPR1 = EXPR2 $( '+' EXPR2(.CLA-1A / .CLA2A) .OUT(,'FAD' , *) .STOA
                /'-' EXPR2(.CLA-1N .OUT(,'FAD',*)/.CLA2A .OUT(,'FSB',*))
                          .STOA)  .,

EXPR2 = EXPR3 $( '*' EXPR3(.CLA-1O/.CLA2Q) .OUT(,'FMP',*) .STOA
               /'//' EXPR3 .CLA2A .OUT(,'FDP',*/,'XCA'/,'FAD','=16488')
                           .STOA
               /'/' EXPR3 .CLA2A .OUT(,'FDP',*) .STOQ)   .,

EXPR3 = PRIMARY $('**' PRIMARY FREEAC *1 .CALL ('.FXP2.('*','*1')')
                           .STOA  ) .,

PRIMARY = .ID / '(' EXPR ')' .,

NEXPR = EXPR2('+' EXPR1 (.CLA-1A .OUT(,'FSB',*)/.CLA2N .OUT(,'FAD',*))
                 .STOA   /'-' NEXPR (.CLA-1A/.CLA2A).OUT(,'FAD' ,*) .STOA
                        / .EMPTY .CLA1N .STOA) .,

    .END
```