1. Credits ........................................................................................................... 1
2. META - A Syntax-Directed Compiler Writing Language .............................. 2
   2.1. What does "syntax-directed mean? .......................................................... 2
   2.2. The Use of a Compiler ............................................................................ 2
   2.3. Writing a Compiler Using Meta .............................................................. 3
   2.4. The Nature of Syntax Descriptions ....................................................... 4
3. The Syntax of a META Program .................................................................... 5
   3.1. Productions .............................................................................................. 5
   3.2. Choices .................................................................................................... 6
   3.3. Term lists .................................................................................................. 7
   3.4. Tests and Actions .................................................................................... 8
4. Meta TEST terms ............................................................................................. 9
   4.1. Single Character Test .............................................................................. 9
   4.2. Multiple Character Test ......................................................................... 9
   4.3. Multiple Character Test with Delimiter Check .................................... 9
   4.4. BLANK test ........................................................................................... 9
   4.5. Assembly Language Tests ..................................................................... 10
   4.6. Invert Pass/Fail ...................................................................................... 11
   4.7. Discard Tokens ...................................................................................... 11
   4.8. Production Call ....................................................................................... 11
   4.9. Nested levels of CHOICES ................................................................... 11
   4.10. Syntax of TESTS .................................................................................. 11
5. META ACTION Terms ..................................................................................... 12
   5.1. Counted Repeat ...................................................................................... 12
   5.2. Message Generating Terms .................................................................... 12
   5.3. Optional CHOICES .............................................................................. 12
   5.4. Repeat Term until Fail ............................................................................ 12
   5.5. CALL Trace Control ............................................................................. 12
6. Output Code Generation ................................................................................ 13
   6.1. Code Generation ACTION terms ........................................................... 13
   6.2. Strings Code Literals ............................................................................ 14
7. OPTIONS and SETUP statements ................................................................ 16
   7.1. FILEID .................................................................................................... 16
   7.2. FILETYPES .......................................................................................... 16
   7.3. Attributes ............................................................................................... 17
   7.4. Compiler Variables ............................................................................... 18
   7.5. Utility Stacks .......................................................................................... 19
   7.6. Keywords ............................................................................................... 19
   7.7. Symbol Value Cells ............................................................................... 20
8. Source Stream Scanner Control ..................................................................... 21
9. Using the META Compiler ............................................................................ 22
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 file and 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 programming, the function of compilers, and the ability to keep separate 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 separately 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 two 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 explanations 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 its 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 separate 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 are groups of characters taken together are are the smallest things that have individual meaning in a language. For example, almost all programming languages use identifiers, 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)
 ; (That's 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 chasing 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 everything 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 \texttt{<termlist>} is:

\begin{verbatim}
TERMLIST = ( \texttt{<test>} : \texttt{<action>} ) $ ( \texttt{<test>} : \texttt{<action>} ) ;
TERMLIST =
\end{verbatim}

The syntax called \texttt{TERMLIST} is

\begin{verbatim}
( \texttt{<test>} : \texttt{<action>} )
\end{verbatim}

Either the syntax of \texttt{<test>} or if not that, then the syntax of \texttt{<action>},

\begin{verbatim}
$ followed by none or more
\end{verbatim}

( \texttt{<test>} : \texttt{<action>} )

choice of the syntax of \texttt{<test>} or \texttt{<action>}

\begin{verbatim}
;
\end{verbatim}

End of the definition of \texttt{<termlist>}.  
A \texttt{<termlist>} ends when the input does not fit the syntax of either \texttt{<test>} or \texttt{<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 ":f" 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:

\begin{verbatim}
NLIT = $ .blank : .nchr $ .nchr ;
\end{verbatim}

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 = "<s1>" ; 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 = "<s1>" ; Generate output code from the pattern given in the string literal. An example: !"\!\subroutine/".

TEXT = ".TEXT" <s1> ; 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 1

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.
5. Assembly Language Tests

Any term that starts with a period and is followed by an identifier is considered a call to a routine. A routine is 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> [ <arg> ]`

The optional arguments are defined by:

`ARG = <numeric literal> | <identifier> | <string literal> | "" | .any`

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

As an example, the test .ASMXAMPL(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
byte 0
even
```

"string of text"
4.6. Invert Pass/Fail

If any test term is preceded 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

\[\text{DTOK} = \sim \sim (\text{<numeric literal> } )\]

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 define 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 \text{<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

\[
\text{TESTS} = \text{<identifier>} \ % \text{production call} \ % \\
\ \
\text{<string literal>} \ % \text{multi-character test} \ % \\
\ \
\text{? (string literal)} \ % \text{test with delimiter check} \ % \\
\ \
\text{? tests} \ % \text{invert pass/fail of next term} \ % \\
\ \
\text{? (integer literal) } \ % \text{discard tokens} \\
\ \
\text{? <identifier> [ are ] } \ % \text{assembly language test} \\
\ \
\text{? chr} \ % \text{single character test} \\
\ \
\text{? ( choices ) } \ % \text{outer level choices as a term} \\
\]
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:

\[ \text{RPT} = \text{"REPEAT" <declare cell identifier>} \]

\[ ( \text{action : test} ) ; \]

5.2. Message Generating Terms

\[ .\text{ERROR} \text{<string literal>} \]

\[ .\text{TEXT} \text{<string literal>} \]

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

5.3. Optional .CHOICES

By enclosing a term or a list of choices seperated by "!" in
brackets, the resulting 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

\[ \text{RF} = \$ <\text{term}> \]

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

5.5. CALL Trace Control

\[ .\text{TRACE} \]

\[ .\text{NOTRACE} \]

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 on 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 separate 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.

Transferring 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:

DEFTTRAN = '!' <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 = '!' '-O' <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
- 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,ADR
`"\li\r0,`*/`
li   `*/
"\mov+\,r0/\mov,`*r3`+`,*/"
mov   NAME,r0
mov   `*r3`,NAME
```

-15-
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 in file 25, out file 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:

```
dec
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:

```
dec
declare print,nref1;
dec
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 value of 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 following 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> $ <kwrd> ; ;
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 nr, dim, tcode, assoc, syequ;
```

would declare that each symbol table entry will have 4 value cells, known as nr, dim, tcode, assoc, and syequ, which might perhaps refer to the number of dimensions, variable type code, and 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, nr, dim) move intbin variable to the nr, dim 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 character to be appended 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-of-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.

- **lflush**: 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)

{test} = {n}

assign variable output stream

0 {s}

variable output from literal string

0 {p}

variable output from code pattern

1{n} {s}

output to deferred stream from literal string

1{n} {p}

output to deferred stream from code pattern

{n}

pop deferred output stream

.PRNDEF(n)

print deferred 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
I [I

JEST TERMS (can pass or fail)

\(<id>\)  invoke production
\(<ch>\)  pass if string literal value is next in stream
\(<cs>\)  as above, but delimiter must be non-blank to pass
\(<term>\)  invert pass/fail of \(<term>\)
\(<ch>\)  discard \(<n>\) tokens
\(<id>\)  discard one token
\(<id>\) (\(<arg>\))  invoke assembly language subroutine
\(<c>\)  test for occurrence of character \(<c>\) next in stream
\(<c>\)  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           |
| .any    | 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
 .atoken queue token to token stack

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

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

 CURSYM = current symbol pointer
CHARACTER CLASS VARIABLES

The character classes are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCUCA</td>
<td>Upper Case Alpha</td>
</tr>
<tr>
<td>CCLCA</td>
<td>Lower Case Alpha</td>
</tr>
<tr>
<td>CCN</td>
<td>Numeric Digit</td>
</tr>
<tr>
<td>CCH</td>
<td>Hex letter A-F or a-F</td>
</tr>
<tr>
<td>CSPCL</td>
<td>Special Characters</td>
</tr>
<tr>
<td>CCA</td>
<td>Alpha upper or lower case</td>
</tr>
<tr>
<td>CCAN</td>
<td>Alpha or numeric digit</td>
</tr>
<tr>
<td>CCHN</td>
<td>Hex digit 0-9, A-F, or a-f</td>
</tr>
<tr>
<td>32</td>
<td>(unused)</td>
</tr>
<tr>
<td>64</td>
<td>(unused)</td>
</tr>
<tr>
<td>128</td>
<td>(unused)</td>
</tr>
</tbody>
</table>

CHARACTER 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
- CLDEL(<char or var>,<class>)  Remove character from class
ATTRIBUTES

\[ .s \langle \text{id} \rangle \quad \text{set global attribute} \]
\[ .r \langle \text{id} \rangle \quad \text{clear global attribute} \]
\[ .t \langle \text{id} \rangle \quad \text{test global attribute} \]
\[ .w \langle \text{id} \rangle \quad \text{set symbol attribute} \]
\[ .w \langle \text{id} \rangle \quad \text{test symbol attribute} \]

VARIABLES (declared)

\[ .c r \langle \text{v} \rangle \quad \text{clear variable to 0} \]
\[ .i nc \langle \text{v} \rangle \quad \text{add 1 to variable} \]
\[ .d e c \langle \text{v} \rangle \quad \text{subtract 1 from variable} \]
\[ .s e t \langle \text{var} \rangle , \langle \text{n} \rangle \quad \text{set variable to value } \langle \text{n} \rangle \]
\[ .m o v \langle \text{fromv} \rangle , \langle \text{tov} \rangle \quad \text{tov=fromv} \]
\[ .m a x \langle \text{v1} \rangle , \langle \text{v2} \rangle \quad \text{v2=max of the two variables} \]
\[ .e q l \langle \text{v1} \rangle , \langle \text{v2} \rangle \quad \text{pass if } \langle \text{v1} \rangle = \langle \text{v2} \rangle \]
\[ .s e n d \langle \text{v} \rangle \quad \text{output decimal value of } \langle \text{v} \rangle \]
\[ .c a d d \langle \text{v1} \rangle , \langle \text{n} \rangle \quad \text{add literal } \langle \text{n} \rangle \text{ to variable } \langle \text{v1} \rangle \]
\[ .v a d d \langle \text{v1} \rangle , \langle \text{v2} \rangle \quad \text{add } \langle \text{v1} \rangle \text{ to } \langle \text{v2} \rangle \]
\[ .v m p r \langle \text{v1} \rangle , \langle \text{v2} \rangle \quad \text{v1} * \langle \text{v2} \rangle \text{ to } \langle \text{v2} \rangle \]
\[ .v l t u \langle \text{v} \rangle \quad \text{pass if } \langle \text{v} \rangle < 0 \]
\[ .e v e n u p \langle \text{v} \rangle \quad \text{round } \langle \text{v} \rangle \text{ up to next even value} \]
\[ .d a d d \langle \text{v16} \rangle , \langle \text{v32} \rangle \quad \text{add 16 bit } \langle \text{v16} \rangle \text{ to 32 bit } \langle \text{v32} \rangle \]
\[ .d m p r \langle \text{v16} \rangle , \langle \text{v32} \rangle \quad \text{multiply 16 bit } \langle \text{v16} \rangle \text{ to 32 bit } \langle \text{v32} \rangle \]
\[ .d n e g (d32) \quad \text{negate 32-bit variable} \]

STACKS

\[ .s p u s h (\text{var}, \text{stk}) \quad \text{Push integer to stack} \]
\[ .s p o p (\text{stk}, \text{var}) \quad \text{Pop stack to integer} \]
\[ \text{fail if stack is full} \]
\[ \text{fail if stack is empty} \]

VALUES of symbols

\[ .v l d (\text{var}, \text{valuename}) \quad \text{set symbol value} \]
\[ .v s t (\text{valuename}, \text{var}) \quad \text{set symbol value to var} \]
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
lfchr char to flush rest of line
lfush 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=mde  -1=NOS
out$ 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 OUTO value in decimal
## generate OUTO value in hexadecimal
#1 generate label unique to production
#2
#3
#4
META II
A SYNTAX-ORIENTED COMPILOR 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 parse structure, and deletes it if found. Backpatch 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 Schmitt, Metcalfe, and Schorre. These papers were presented at the 1961 National A.C.M. Convention in Denver, and represented the activity of the Working Group on Syntax-Directed Compilers of the Los Angeles SIGFPLAN. 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 and Baslton. All of these methods can be traced back to the mathematical study of natural languages, as described by Chomsky.

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:

LOGICAL VALUE = '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 . ENDT. For example:

SUBSECONDARY = '*' PRIMARY / . ENDT .

SECONDARY = PRIMARY('*' PRIMARY / . ENDT) .

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

SECONDARY = PRIMARY('*' SECONDARY)

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 META II 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 some embedded 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:

D1.3-1
The syntax equations which define this class of expressions are as follows:

\[
\begin{align*}
E_1 & = A + B \\
E_2 & = A + B + C \\
E_3 & = (E_2) \\
E_4 & = E_1 + E_2 \times E_3
\end{align*}
\]

The symbol \( E \) is an abbreviation for expression. The

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:

\[ S = \text{SEQ} \]

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

\[
\begin{align*}
E_1 & = A \\
E_2 & = E_3 \times E_4 \\
E_3 & = (E_2) \\
E_4 & = E_1 + E_2 \times E_3
\end{align*}
\]

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 VALCOIL I and VALCOIL 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 the first column 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:

\[
\begin{align*}
E_1 & = .ID .OUT(\text{'\text{'E'\text{'1}}') .
E_2 & = E_1 \times E_3 (\text{'\text{'E'\text{'1}} . .)
E_3 & = (E_2)
E_4 & = E_1 + E_2 \times E_3
\end{align*}
\]

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:

\[
\begin{align*}
\text{.ID .LABEL}
\end{align*}
\]

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 .E' .EX .OUT(\text{\text{'E'\text{'P'\text{'F'\text{'A'\text{'R'}}})
STATEMENT .ELSE .OUT(\text{\text{'E'\text{'S'\text{'T'\text{'E'\text{'M'}}}) .LABEL .1
STATEMENT .LABEL .2
```

The op codes BPF and B are orders of the VALCOIL 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.

```
VALCOIL - A Simple Compiler Written in META II
```

Now we are ready for an example of a compiler written in META II. VALCOIL is an extremely simple language, based on MOLCOM40, which has been designed to illustrate the META II compiler.

The basic information about VALCOIL I is given in figure 1 (the VALCOIL I compiler written in META II) and figure 2 (order list of the VALCOIL machine). A sample program is given in figure.

After each line of the program, the VALCOIL commands which the compiler produces from them are shown, as well as the absolute interpretation language produced by the assembler. Figure 3 output from the sample program, let us study this 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 the control goes there first. The equation for MARY is similar to that of \( E_1 \) in our previous example, but here numbers are recognized as produced with a "load literal" command. This is what makes the output shown earlier so \( E_1 \) except for recognition minus extraction. The equation E XP defines the relational operator "equal", which produces a value "
Notice that this is the same statement as the one in AJOI. So, it is not possible to choose a "true" or "false" answer.

The condition is that the equal and inequality operators are used in the statements. The "true" and "false" answers are the same as in the original program. The "true" answer is the one in which the producer is not equal to the consumer. The "false" answer is the one in which the producer is equal to the consumer.

For the META II compiler, which is an interpreter, the first program is used to produce another program. This program is used to produce the final result.

The programs for the META II compiler, which is an interpreter, are written in the same language. The META II compiler is written in the language of the program.
VALOOL I machine, and which require some explanation. In the VALOOL 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 number 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 addresses 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.

A 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 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Arguments in reverse order</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXXXXX</td>
<td>XXXXXXX</td>
<td>XXXXXXX</td>
</tr>
<tr>
<td>b</td>
<td>Word of one blank character to mark the end of the arguments.</td>
<td></td>
</tr>
<tr>
<td>Body. Branch commands cause control to go around data stored in this area. Ends with a &quot;return&quot; command.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Illustration 1

Storage Map for VALOOL II Procedures

XXX Address of procedure XXX Stack before executing call instruction

Illustration 2

Map of the Stack Relating to Procedure Calls

number of arguments in the stack does not correspond to the number of arguments in the procedure, an error is indicated. The "flag" in the stack works like this. In the VALOOL II machine there is a flag register. To set a flag in the stack, the contents of this register is put on top of the stack, then the address of the word above the top of the stack is put into the flag register. Initially, and whenever there are no flags in the stack, the flag register contains blanks. At other times it contains the address of the word in the stack which is just above the uppermost flag. Just before a call instruction is executed, the flag register contains the address of the word in the stack which is two above the word containing the address of the procedure to be executed. The call instruction picks up the arguments from the stack, beginning with the one stored just above stack

same

... same old old

... same

takes

Also, The

as fi

A9

A5

A5

A5

D1.3-4
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 "ENTER" primarily because truncation takes fewer machine instructions to implement. Also, truncation seems to be used more frequently. The procedure ENTER can be defined in VALOOL II as follows:

```
.PROCEDURE ENTER(X) ,
    .IF 0 <= X THEN WHOLE (X) .ELSE
    .IF WHOLE(X) = X THEN X .ELSE
    WHOLE(X) - 1
```

The "for statement" in VALOOL 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 1 = 0 .STEP 1 .UNTIL N .DO
    (statement)
    A91 GRT
    LD 1
    FLP
    RBF A92
    LDL 0
    GRT
    B A99
    A92 LDL 1
    ADS
    A93 RDR
    LD N
    LEG
    RBF A94
    (statement)
    REST
    B A91
```

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

```
Flow chart of the "for statement"
given in figure 15:
```

Figure 7 is a listing of the VALOOL II compiler written in META II. Figure 8 gives the order list of the VALOOL II machine. A sample program to take a determinate value 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 ALOOL and COBOL can be implemented without backup is pointed out by various people, including Conway, 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 "&" or "&". 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 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 has implemented LOGIX, 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 have implemented META 3 on the IBM 7094, with the goal of producing an ALOOL 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 same META 4 (pronounced metaphor).

References

THE VALCO 3 COMPILED LISTED IN META II LANGUAGE

FIGURE 2

JANUS PROGRAM

PUBLIC 10.0, OUT1:0.0
PUBLIC @START, @END, @RESULT, @SUBTRACT, @MULTIPLY

BEGIN
PUBLIC @START

A = 10.0, B = 5.0

A = B + A

A = A

B = B - A

B = B

A = A * B

A = A

B = B / A

B = B

END

A PROGRAM AS COMPILED FOR THE VALCO 3 MACHINE

FIGURE 3

ORDER LIST OF THE VALCO 3 MACHINE

FIGURE 4

MACHINE CODES

LD AAA LOAD
LDL NUMBER LOAD LITERAL
ST AAA STORE
ADD ADD
AND AND
SUB SUBTRACT
MUL MULTIPLY
DIV DIVIDE
SUB "ADD" SUBTRACT
JMP JUMP
JSR JUMP CALL
BRA BRANCH
BST STRING EDIT
PRINT PRINT
HALT HALT

OUTPUT FROM THE VALCO 3 VALCO-2 LISTED IN FIGURE 4

FIGURE 5

BLK O.C. SCHORRE - ACM PROCEEDINGS 1964
THE META IS COMPILED HHITEN IN ITS OWN LANGUAGE

---

[Code listing and diagrams for the meta language compiler are shown here.]

---

D.V. Schorr - ACM-Proceedings 1964
EXAMPLE PROGRAM ON COBOL II

DATA DIVISION
  DECK
    PROCEDURE DIVISION
    MAIN
      CALL DETERMINANT
      END-MAIN
  ENDDATA

PROCEDURE DETERMINANT
  DETERMINANT:  DECIMAL 0

PROCEDURE DETERMINANT
  DETERMINANT:  DECIMAL 0

READ 0, G, J, ...; OTHER A, B, C, D...
REAL PRODUCT, FACTOR, TEMP, A = 2, D = 1
PRODUCT = 1
FOR I = 0, STEP I, UNTIL B = 0
  IF A > B, PRINT = A, B
  PRODUCT = PRODUCT * A
  A = A + 1
  GO TO 2
END-MAIN

END OF PROGRAM
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 backtracking based on the compiler written for the IBM 1401 by Val Schorr. Each statement of the language is scanned 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 interpreted with the syntactical 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 is a simplification of the META II compiler developed by D. V. Schorr on an IBM 1401, but it is modified to facilitate direct generation of sequential code rather than polished-like code.

The META-3 compiler constructs a series of tests and references 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, testing flags and so forth. There are two major types of test. One is to test internal status such as the value of a variable, the other is to test the input stream for the occurrence of 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-3 compiler generates the code to test this value after every test and either proceeds, if true, or, if false, tries to return the value false to the caller. Since anything tested for and found in data is found in data, 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 the bad syntax. The discussion of the syntax equations for META-3 written in META-3 will allow the usage and definition of the basic syntax elements. For a further discussion of the basic syntax elements or references on the subject see Schorr's paper in this volume.

Each syntax equation begins by naming the construct which is defining, and code with a semantic (writes "*"). The definition is a series of tests and actions, which may be grouped by parenthesis. A string is quoted (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 in 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 non-recognizable character, usually a blank. The first 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. "END" 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 .2242) 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, operand, and variable. Fields are separated by commas, and cards are terminated by semicolons. There are three forms which each argument may take:

a) string to be inserted literally
b) * indicating the uppermost element of the stack
c) "n" indicating the nth label stack, each of which has a unique constant value at each usage of each statement.

"CALL(...)" is equivalent 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
Represents a 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")*
Output 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 executed).

*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 .ID .OUT("ENTRY")*.ST .END

It may be expressed in Backus Normal Form as:

program := SYNTAX (identifier <read> .END <stmt> := <stmt> <stmt> <stmt> <stmt> ) Empty

D1.5-1
The operations in META-3 continue

ST = 1D .OUT(·"PXA",/;·"CALL",·"PUSH")

This is the beginning of the definition of a
statement and a label. Both the statements
are in the same instruction, but they are
followed by labels.

/" EXI " CASE (.·"POP")
The identifier must be followed by an equal symbol
and an EXI (see below) and terminated by a
label. At this point a call of .OUT, .POP is output.
The routine .PUSH and .POP hands the
recursion.

EX1 = EX2 & /" .OUT(·"ZET",·"TEST",·"TRA",·"1") EXI"

/" .OUT(·"1, NULL")
An expression can be defined as an expression
variable. It follows a sequence (which may be
empty) of label, at which point output a test for
the truth of the previous expression, which
must be an internal transfer of control to the label.

EX2 = ( TEST OUTPUT, .NZT",·"TEST",·"TRA",·"A") ACTION)

" TEST OUTPUT, ·NZT",·"TEST",·"TRA",·"A") ACTION)
This expression is made of a number of
tests or actions. This is a statement in which
the rest of them are skipped. If any of the
others in any test .DIAG receives control.

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

/" .ID (·"CALL",·"JORT")

or the action "ID" which is then called on a
routine. .JORT is generated.

/" EXI "

or, left parenthesis followed by an EXI
followed by a right parenthesis.

/" STRING CALL (·"CAPR")

or, a string is quoted whose location is inserted
into the stack and then output as an argu-
ment to .CAPR.

/" STRING CALL (·"STRT")

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

/" CLA (.·"DIGIT ALPHABETIC")

or, the word .CLA followed by a minus sign
and a digit and a letter both of which are placed
in the stack on they are found by external

/" CLA (.·"DIGIT")

These two characters are output as argu-
ments of a call on routine..CLAD by placing the letter
in the "1" label stack and referencing it in the
.CALL statement.

/" DIGIT ALPHABETIC (·"CALL",·"CLAD")

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

/" ALPHABETIC CALL(·"MINS")

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.

/" MINS (.·"DIGIT CALL (.·"MOVE")

Or, an asterisk followed by a digit which is
compilation as an argument to routine .MOVE.
This is in the routine which moves the digit
and .NULL to the opened stack and the label
stack.

/" .DIGIT CALL (.·"MOVE")

Or, the asterisk could be followed by a minus
sign and a digit which is given as an argu-
ment to move the "1" stack to the opened
stack.

/" ALPHABETIC CALL (·"STAR")

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.

/" TEST (·"SET")

Or, the final thing which a test may be is
test followed by a sequence which is used as an
argument to the generated call on .SET.

/" .ACTION OUTPUT

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

/" .EMPTY .OUT (.STL",·"TEST")

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

/" .OUT (.STL")

Or, a dollar sign, at which point the label is
"1" output, followed by a test (defined above)
.OUTPUT (.ZET",·"TEST",·"TRA",·"STL")" TEST)
after which the test is set if it will be rep-
ated, otherwise, .TEST is set non-zero to
indicate a retest.

/" .STO" ALPHABETIC .CALL (.STOR")

Or, .STO followed by a letter which is com-
plied on as an argument to .STOR.

/" .ALPHABETIC .CALL (.PLUS")

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

/" .S" ALPHABETIC .CALL (.SET",·"S")

Or, finally, an action may be S-followed by a
letter which becomes, at a time in which it is
an argument to .SETS (which sets the mode list
for later testing with .S).

/" OUTPUT (·"1, NULL")

As output is defined to be

/" .OUT (."1")

.OUTPUT followed by an (OUT1 in parentheses
.CALL (."FELD") .CALL (."LITQ 01324193700")
.CALL (."FELD")

Or, .CALL which generates the same instruc-
tions as .OUT .CALL (·")...

/" .OUT (·"OUT2")

or, .OUT followed by a digit which is
COMPLED as a sequence (which may be empty)
of OUT1's after which a call on .PUBG is generated.

/" .ERITE (."OUT")

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

OUT2 = OUT2A & (·"CALL",·"PUBG") OUT2A .CALL

(·"PUBG")

An OUT2 is defined to be by an OUT2A followed
by a sequence (which may be empty) of OUT1
(which point a call to .PUBG in output)
followed by OUT2A's at the end .PUBG is
called.

OUT2A = .OUT (."1") .CALL (."FELD") .OUT1)

An OUT2A is defined to be a sequence
(which may be empty). An .OUT1 followed by a
sequence which may be empty and an internal
output a call on .FELD followed by sequence
(which may be empty) of OUT1A.
OUTI = "(DIGIT CALLU..GENRE y)"

As OUTI is defined to be either an asterisk followed by a digit, in which case GENRE is called with the digit as an argument,
/ EMPTY CALLU.COP() // or else, for an asterisk alone, a call on COP() is output.
/ STRING CALLU.LIT() + x \\

Or, finally, an OUTI 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 ISMAP assembler assumes
that any undefined symbol will be defined as an entry point to some other task at load time.
This rather naive assumption on the assembler part allows operations to be added at will with the understanding
that if the added routine is actually only an action the compiler
will treat it as a test, and execute cell .TEST on return from the routine, and had better find it non-zero (or true)
at that point if any error messages are to be avoided and compilation is to continue.

The Push-down Operated Stack

The meta-linguistic elements * 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
an output imperative, by the FIFO, or by entering subroutine REMOVE which may be called either from a syntax equation
or from a build-coded routine.
This stack is extended by allowing copies to be freely made from the * stack to any one of the four local safe
All other operations 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 METAV is the
combined push-down stack and first-out list. The operations
FEE, F1, FO, and FUM are used to address the elements
being inserted by F1 and removed by FO. FEE is used
to push the list down and insert a label 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 in the list is that of a number of
superimposed FIFOs (linear queues). F1 removes the
uppermost element of the operand stack and inserts it into the FIFO list as the last element.

F0 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
F0 will return false and pop to the top of the
stack. FEE starts a new list, marking the top of the previous
one. That is it pushes down the previous 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.

<table>
<thead>
<tr>
<th>Operated stack</th>
<th>Operation</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B G D E</td>
<td>FI</td>
<td>empty initially</td>
</tr>
<tr>
<td>A B G A</td>
<td>FI</td>
<td>E</td>
</tr>
<tr>
<td>A B G</td>
<td>FE</td>
<td>GAE</td>
</tr>
<tr>
<td>A B</td>
<td>FI</td>
<td>GAE</td>
</tr>
<tr>
<td>A B</td>
<td>FO</td>
<td>GAE</td>
</tr>
<tr>
<td>B A</td>
<td>FO</td>
<td>GAE</td>
</tr>
<tr>
<td>B A</td>
<td>(return false)</td>
<td>FUM</td>
</tr>
<tr>
<td>B</td>
<td>(output CALL A(T,D,E))</td>
<td>empty</td>
</tr>
</tbody>
</table>

Invertible rearrangements and resequencing are possible using
this list since anything out wanted now can be filed 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 times by doing alternate F0s and
FIs and rolling up without changing the length of the area
(this action reminds me of a tracked vehicle), while
determining the parameters necessary for storage allocation.

The Symbol Table

This routine stores and examines symbols gives to it.
Each symbol may have any of 26 arbitrary attributes, represented
as a through Z. These properties are given to the
routine by oring a property into the attribute register. For example +R adds the property R to those properties already
in the attributes register. The meta-language primitive CLEAR
erases all the properties to false. The meta-language primitive
SET unconditionally 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 treated 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, semantically 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 treated 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 LOC
AL returns true if, in the identifier listed 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 addressing, a Q regis-
ter useful for divisions, an I register and a L register,
both used for logical operations, an R register used for dou-
ble precision work, and the N register which is a negative
A register. It is assumed that these two registers cannot both
contain information at the same time.
The safeguarding of the contents is ensured by the imperative .STOA where a 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 where n is the depth of the "stack" that the average reference is to be taken from and a 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 preceded by -. In this case the register exchange is performed only if it is in a pure exchange; that is, the requested operand is already in the register.

EXAMPLE:

```
.SYNTAX EXP
EXPR = EXPR FREEAC
EXPR = EXPR S ("*" EXPR (CLA1A/CLA2A)
.OUT("ADD","!" .STOA)

EXPR = PRIMARY $ ("** PRIMARY,CLA-1Q/CLA2O)
.OUT("MPY","!" .STOA)

PRIMARY = "Y" EXPR (Y / JD ,)
.END
```

Given 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
STQ .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 temporary in use in any block. At the end of the block the verb TPUSH will generate the instruction:

```
.TBSS n
```

where n is the number of temporary used.

A more complex example of the use of some of the features of METAC is the listing of the syntax equations for CODOL in the appendix. CODOL is a minimal compiler, designed first 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 REAL NUMBER as a PRIMARY. This routine is a hand-coded one designed for the ALCOL 60 compiler now under development using the successor to META-3, META-4.

Acknowledgments

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

References

### Appendix A

**Instructions generated by register exchanges**

<table>
<thead>
<tr>
<th>Source</th>
<th>A</th>
<th>N</th>
<th>Q</th>
<th>Destination</th>
<th>L</th>
<th>I</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>CHS</td>
<td>XCA</td>
<td>LDQ =0</td>
<td>XCA XCL PAI</td>
<td>STO</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Illegal</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>XCA</td>
<td>XCA CHS</td>
<td></td>
<td>XCA LDQ =0</td>
<td>XCL XCL PAI</td>
<td>STU</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>CHS</td>
<td>XCA</td>
<td></td>
<td>XCA XCL XCA XCL PAI</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>XCL XCA XCL XCL CHS</td>
<td>XCL</td>
<td>XCL XCL LDQ =0</td>
<td></td>
<td>PAI</td>
<td>SLE</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>PIA</td>
<td>PIA XCL PIA</td>
<td>PIA PIA XCL XCL XCA GIS LDQ =0</td>
<td>PIA</td>
<td></td>
<td>ST</td>
<td></td>
</tr>
</tbody>
</table>

Storage: CLA CLS LDQ DLH CAL LDI

Note: The transfers between L and any of A,Q,R, or N are for completeness only, there are convenient instructions for this.

Note: The transfer to SLE or STU are Illegal, there being no convenient instruction for this.
### Appendix D

#### Subroutines used and their Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage</th>
<th>Statement</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPP</td>
<td></td>
<td>ST</td>
<td>Saves location of caller for recursion</td>
</tr>
<tr>
<td>PUSH</td>
<td></td>
<td>ST</td>
<td>Recursive return</td>
</tr>
<tr>
<td>TEST</td>
<td>any</td>
<td>EX1, EX2</td>
<td>Non-zero if true, zero if false.</td>
</tr>
<tr>
<td>DIAG</td>
<td>any</td>
<td>TEST</td>
<td>Prints stack, input stream and nasty message about bad syntax.</td>
</tr>
<tr>
<td>IDNT</td>
<td>.ID</td>
<td>TEST</td>
<td>Test for an identifier in the input stream, places first six characters in the stack if found.</td>
</tr>
<tr>
<td>COMP</td>
<td>&quot;XYZ...&quot;</td>
<td>TEST</td>
<td>Tests for a string in the input stream returns true on match.</td>
</tr>
<tr>
<td>STRT</td>
<td>.STRING</td>
<td>TEST</td>
<td>Tests for any string in the input stream, outputs it as:</td>
</tr>
<tr>
<td>CLAD</td>
<td>.CLAdx or .CLAdx</td>
<td>TEST</td>
<td>and places .Zmm in the stack.</td>
</tr>
<tr>
<td>MINS</td>
<td>-x</td>
<td>TEST</td>
<td>Entry point to register manipulator for register loads and exchanges.</td>
</tr>
<tr>
<td>MOVE</td>
<td>&quot;x&quot; or &quot;x-x&quot;</td>
<td>TEST</td>
<td>Used to test input string and compare it with the symbol table.</td>
</tr>
<tr>
<td>STAR</td>
<td>&quot;Z&quot;</td>
<td>TEST</td>
<td>Moves single elements from the stack to the label stack and vice-versa.</td>
</tr>
<tr>
<td>SETT</td>
<td>.Tx</td>
<td>TEST</td>
<td>Used to compare the stack with the symbol table.</td>
</tr>
<tr>
<td>STOR</td>
<td>.STOa</td>
<td>ACTION</td>
<td>Used to test the mode flag.</td>
</tr>
<tr>
<td>PLUS</td>
<td>+x</td>
<td>ACTION</td>
<td>Tells the register manipulator to hang onto the contents of x.</td>
</tr>
<tr>
<td>GENR</td>
<td>&quot;n&quot;</td>
<td>OUT1</td>
<td>Set the symbol table.</td>
</tr>
<tr>
<td>DIGIT</td>
<td>&quot;n&quot;</td>
<td>OUT1</td>
<td>Set the mode flag to X.</td>
</tr>
<tr>
<td>ALPHAB</td>
<td>x</td>
<td>TEST</td>
<td>Begins a new field on output.</td>
</tr>
<tr>
<td>CHARAC</td>
<td>x or n</td>
<td>ACTION</td>
<td>Moves a fixed string into the output stream.</td>
</tr>
<tr>
<td>CLEAR</td>
<td></td>
<td>ACTION</td>
<td>Ends a card image.</td>
</tr>
<tr>
<td>TPUH</td>
<td></td>
<td>ACTION</td>
<td>Writes an error message.</td>
</tr>
<tr>
<td>TOPP</td>
<td></td>
<td>ACTION</td>
<td>Moves the label from the &quot;n&quot; label stack to the output stream.</td>
</tr>
<tr>
<td>USE</td>
<td></td>
<td>ACTION</td>
<td>Moves the stack to the output stream and pop it up.</td>
</tr>
<tr>
<td>USEPOP</td>
<td></td>
<td>ACTION</td>
<td>Moves one character of the specified type into the stack.</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>ACTION</td>
<td>DIGIT and ALPHAB may return false, CHARAC never.</td>
</tr>
<tr>
<td>SET</td>
<td></td>
<td>ACTION</td>
<td>Remote attribute registers.</td>
</tr>
<tr>
<td>REMOVE</td>
<td></td>
<td>ACTION</td>
<td>Marks the beginning of a block of temporary.</td>
</tr>
<tr>
<td>FEE</td>
<td></td>
<td>ACTION</td>
<td>Ends a block of temporary and allocates storage to them.</td>
</tr>
<tr>
<td>FUM</td>
<td></td>
<td>ACTION</td>
<td>Begins block of separate code.</td>
</tr>
<tr>
<td>REALN</td>
<td></td>
<td>ACTION</td>
<td>Ends block of separate code and returns to previous block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACTION</td>
<td>Defines a symbol in the stack with the properties in the attribute registers, and puts it in the symbol table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACTION</td>
<td>Deletes the top of the stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACTION</td>
<td>Reference the FIFOlist (see text).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST</td>
<td>Try to get a double-precision floating-point number from the input stream.</td>
</tr>
</tbody>
</table>
APPENDIX C

7094 META-COMPILER COMPILED BY ITSELF,

*SYNTAX PROGRAM
OUT1 =
  "+: ( DIGIT CALLI "GENR( 1 )")
  / EMPTJ CALLI "COPG( 1 )")
  / STRING CALLI "LITG(1 " )")
  ...
OUT2A =
  $ OUT1 $ ( / CALLI "FELD( 1 )" ) $ OUT1 )
OUT2 =
  
  OUT2A $ ( CALLI "PUBG( 1 )" ) OUT2A ) CALLI "PUBG( 1 )"
OUTPUT =
  "OUT" ( OUT2 )
  / CALLI "FELD( 1 )" ) CALLI "LITG(0231434377001)"
  / CALLI "FELD( 1 )")
  $ OUT1 $ ( CALLI "PUBG( 1 )"
  / ERITE ( ( DIGIT $ OUT1 ) CALLI "ERITE(# 1 )"
ACTION =
  OUTPUT ( "EMPTY" ) OUT ( "STL" ) "TEST"
  / "$" OUT ( "NULL" ) "TEST"
  / OUT ( "ZET" ) "TEST" / "TRA" * 1 / "STL" "TEST"
  / "STOR" ALPHABETIC CALLI "STOR0H( 1 " )""
  / "ALPHABETIC CALLI "PLUS( 1 " )""
  / "$" ALPHABETIC CALLI "SET0H( 1 " )"
  ...
TEST =
  ID CALL ( " )" / "ID" CALL ( "IDNT" )
  / ( EX1 )"
  / STRING CALLI "CMPP( 1 " )"
  / STRING CALLI "STR""
  / CLA ( -1 DIGIT ALPHABETIC * 1
  / CALLI "CLAI(# 1 " )"
  / DIGIT ALPHABETIC ( CALLI "CLAI(# 1 " )"
  / "$" DIGIT CALLI "MOVE1( 1 " )"
  / "$" DIGIT CALLI "MOVE( 1 " )"
  / ALPHABETIC CALLI "STAR( 1 " )"
  / "$" ALPHABETIC CALLI "SET( 1 " )"
  ...
EX2 =
  / TEST OUT ( "NZT" ) "TEST" / "TRA" * 1 ) / ACTION
  $ ( TEST OUT ( "NZT" ) "TEST" / CALL" "DIAG"
  / ACTION ) OUT ( "ZET" ) "NULL"

  ...
EX1 =
  EX2 $ ( / OUT ( "ZET" ) "TEST" / "TRA" * 1 )
  EX2 $ ( OUT ( "NULL" )

  ...
ST =
  ID OUT ( "PXA" ) "*4" / CALLI "PUSH"
  / EX1 ( CALLI "POPP"

PROGRAM =
  SYNTAX ID OUT ( "ENTRY" ) *
  $ ( "ST" OUT ( "ENTRY" ) )
  $ ST "END"

DL.5-7
Appendix D

CODOL COMMON DEMONSTRATION ORIENTED LANGUAGE

SYNTAX PROGRAM

PROGRAM OUT('********','SAVE') TPUSH SEGMENT OUT('RETURN','********')
$ (10 ID = 1 OUT('1','SAVE') SEGMENT OUT('RETURN','*1'))
TPUSH

SEGMENT DECLARATION*** SIDECRACK***ST*$1,1 ST $1)

DECLARATION = 'REAL' OUT('USE','STOR') CLEAR +R ID SET
OUT('PZE') $(*1) ID SET OUT('PZE')

ST = *ID CLEAR $1 SET *STRING OUT('EQU',*1)

ST = **ID OUT('TRA',**1) ST

GO TO ID OUT('TRA',*)

'CALL ID FEE (*1 EXP F R EAA C F1 $(*1 EXP F R EAC F1))

'FUM

'SET FEE .ID F1 $(*1).ID F1) *** EXP .CLA1A

FO OUT('STO',*1) $IFO OUT('STO',*1)

IF EXP CLA1A ('PLUS' OUT('T11',*1) 'MINUS' OUT('TPL',*1))

'ZERO' OUT('TN2',*1) NON ZER0' OUT('T2E'

'*1) ST OUT('NULL')</n

'ALTER ID TO ID OUT('AXT',*1).4'/SXA',**1)

'PRINT ID S CALL(*1 FWRD('UNOS',**1))

$(*1 EXP CLA1A OUT('TSX',*1 FCNV**4))

'READ ID S CALL(*1 FRRD(' UNOS',**1))

$(*1 ID OUT('TSX',*1 FCNV**4 STO'),*1)

EXPR = '**NEXPR / ('1 '*/EMPT Y) EXP1 ,''

EXPR = 'EXP1 = EXP2 $(** EXP2 CLA1A / CLA2A) OUT('FAD',**) .STOA

/* EXP2 = EXP2 CLA1A OUT('FAD',*) .STOA

EXP2 = EXP3 $(** EXP3 CLA1A CLA2A) OUT('FAD',**) .STOA

EXP3 = EXP3 CLA2A OUT('FAD',**) .STOA

EXP3 = PRIMARY $(** PRIMARY FREEAC $1 CALL(*1 FXP2(**1)*/STOA))

PRIMARY = .ID / ("EXPR") **

NEXPR = EXP2 $(** EXP1 CLA1A OUT('FAD',*) .STOA

END