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, writes a relocatable output text file, which describes the assembly statements in a numeric form, as if the program started at address 0000 in memory.

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

The process may be pictured as:

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

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

EDIT >> COMPILE >> LINK >> RUN

2.3. Writing a Compiler Using Meta

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

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

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

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

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

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

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

5) You must document your compiler and routines at 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 explainations of why something was done the way it was.

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

2.4. The Nature of Syntax Descriptions

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

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

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

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

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

In META you could describe this with:

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

Which translates back into english as:

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

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

3. The Syntax of a META Program

META is a recursively defined language. Each part of it) is defined using smaller pieces. When we get to the small pieces, we find that many of them are defined by using the "higher level" pieces. It is like a cat 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 everyting inside them is to be considered a single term. The $ indicates that the next single term is to be repeated as many times as it is matched.

```
CHOICES =               The syntax called CHOICES is defined as

<termlist>              The syntax <termlist>

$                       followed by none or more

(                       of the following group:

  '!                    The character !

  <termlist>            The syntax <termlist>

)                       (end of the group to be repeated)

;                       (end of the definition of CHOICES)
```

3.3. Termlists

A definition of <termlist> is:

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

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

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

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

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

3.4. Tests and Actions

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

Some example tests are:

```
TEST1 = '' <chr> ;        Test for the existence of a single
                          character. We used this above with
                          '= to test for an equal sign

TEST = '" <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:
                          !"\b1\subroutine/".

TEXT = ".TEXT" <s1> ;     Send the string literal to the console
                          as a message
                          .TEXT "PL0 Compiler V1.0".
```

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

4. Meta TEST terms

4.1. Single Character Test

SCTEST = '' chr ;

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

4.2. Multiple Character Test

MCTEST = <string literal>

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

4.3. Multiple Character Test with Delimiter Check

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

4.4. BLANK test

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

.BLANK

This is an example of an assembly language test reference.

4.5. Assembly Language Tests

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

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

The optional arguments are defined by:

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

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

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

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

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

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

4.6. Invert Pass/Fail

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

4.7. Discard Tokens

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

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

4.8. Production Call

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

4.9. Nested levels of CHOICES

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

4.10. Syntax of TESTS

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

5. META ACTION Terms

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

5.1. Counted Repeat

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

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

5.2. Message Generating Terms

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

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

5.3. Optional CHOICES

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

5.4. Repeat Term until Fail

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

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

5.5. CALL Trace Control

```
.TRACE
.NOTRACE
```

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

6. Output Code Generation

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

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

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

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

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

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

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

6.1. Code Generation ACTION terms

The form of the direct output ACTION term is:

DCODE = '! <string code literal>

The form of a deferred output ACTION term is:

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

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

To transfer code from a deferred output stream, use:

DEFTRAN = '^ <numeric literal>

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

The form used to select the switched output stream is:

SWSEL = '! '= <numeric literal>

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

To generate code to the switched output stream, use:

SWCODE = '! '0 <string code literal>

6.2. String Code Literals

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

- \ Tab to next assembly field
- / end the line of assembly code and send it to the output stream
- 'c copy the next character exactly. This is used to output characters that have other meanings.
- * output the top token and remove it from the token stack.
- + output the top token, but leave it on the token stack.
- #0 Generate a decimal number for the value in OUT0.
- #n Generate a label unique for this production call. There are four such labels available for each production iteration.

All other characters are copied exactly as they appear.

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

```
.       !"\pshr\r0/"
        pshr        r0

.       !"\li\r0,*/\mov\r0,*/"
        li          r0,NAME
        mov         r0,ADRS

.       !"\li\r0,'"'*'"/"
        li          "*"

.       !"\mov\+,r0/\mov\'*r3'+,*/"
        mov         NAME,r0
        mov         *r3+,NAME
```

7. OPTIONS and SETUP statements

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

7.1. FILEID

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

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

7.2. FILETYPES

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

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

As an example:

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

is used by the META compiler itself.

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

7.3. Attributes

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

To declare an attribute, use the .attribute statement:

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

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

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

Each attribute becomes an assembler equ statement:

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

translates into:

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

To use global attributes, you use the following terms:

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

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

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

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

7.4. Compiler Variables

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

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

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

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

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

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

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

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

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

```
.gend(var)
```

GEND generates a decimal number equal to the 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 folowing terms:

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

7.6. Keywords

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

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

kwrd = .achr $ .anchr ;
```

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

An example is:

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

7.7. Symbol Value Cells

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

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

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

For example:

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

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

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

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

for example:

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

8. Source Stream Scanner Control

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

eolchr — This cell holds the chaaaracter to be 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-fo comment character appears.

cmtend — This character ends a comment. If comments are handled by a statement type such as REM in BASIC, set cmtchr and cmtend to 0 to disable comments.

lflchr — This character appearing in the source stream will flush to the end of the line and get the next source line as if it were on the same physical line of text.

lflush — This switch causes the line flush action. If your program decides to ignore the rest of an input line, set this variable to 1.

symuc — If this switch is not zero, all characters except those accessed through .ANYC will be converted to upper case.

smode — This switch controls string mode. When it is non-zero, comments controlled with cmtchr and cmtend are temporarily disabled, so that those characters may be used in strings.

colcnt — This cell holds the column number of the character last accessed, starting with 1. If it is zero, the next character will be the first character on a line.

In addition there is one test term provided:

.NEOL

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

9. Using the META Compiler

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

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

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

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

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

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

```
META  =program
```

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

```
META program
```

will use program.rel and program.met

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

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

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

META 3.5 QUICK-REFERENCE SUMMARY

STRUCTURES

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

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

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

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

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

OPTIONS

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

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

ACTION TERMS (NOTEST)

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

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

TEST TERMS (can pass or fail)

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

TOKEN BUILDING TERMS

.achr	alpha character builds
.anchr	alpha or digit ok
.nchr	digit ok
.hchr	hex digit ok
.anyc	any character ok
.untokn	remove char last appended to build buffer
pvchr	= character accepted by test
pvnum	= 0 thru 9 value of last chr if digit and 10 thru 35 for A thru Z
.mtoken('c)	if next chr is "c" then append it
.itoken('c)	append the character "c"
.kwchk	pass if token not a keyword if it is, return token to instream & fail
.qtoken	queue token to token stack
.fvmbl	pass if token is previously defined set CURSYM
.asymbl	add (define) token as new symbol set CURSYM
.rsymbl	reference symbol from CURSYM for attributes values, etc.
.symscn	initialize symbol table scan
.nxtsym	append next symbol to build buffer normally followed by .qtoken sets CURSYM
CURSYM	current symbol pointer

CHARACTER CLASS VARIABLES

The character classes are:

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

CHRACTER CLASS OPERATIONS

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

ATTRIBUTES

```
.s (<id>)    set global attribute
.r (<id>)    clear global attribute
.if (<id>)   test global attribute

.as (<id>)   set symbol attribute
.ar (<id>)   reset symbol attribute
.aif (<id>)  test symbol attribute
```

VARIABLES (declared)

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

.dadd(<v16>,<v32>)     add 16 bit v16 to 32 bit v32
.dmpy(<v16>,<v32>)     multiply 16 bit v16 to 32 bit v32
.dneg(d32)             negate 32-bit variable
```

STACKS

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

VALUES of symbols

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

SCAN CONTROL

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

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

OTHER STANDARD VARIABLES

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

CODE GENERATION ELEMENTS

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