NAT'L INST. OF STAND & TECH R.I.C.
A11105 851781
NBS
PUBLICATIONS

NBSIR 80-2182

M7: A General Pattern Matching Facility Users Manual

A. R. Marriott
G. H. Skillman
S. B. Salazar
W. T. Hardgrave

Application System Division
Center for Programming Science and Technology
Institute for Computer Sciences and Technology
U.S. Department of Commerce
National Bureau of Standards
Washington, DC 20234

January 1981

QC
100
.U56
80-2182
1981
c. 2

U.S. DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS

NATIONAL BUREAU
OF STANDARDS
LIBRARY
MAY 22 1981

NBSIR 80-2182

M7: A GENERAL PATTERN MATCHING FACILITY USERS MANUAL

A. R. Marriott
G. H. Skillman
S. B. Salazar
W. T. Hardgrave

Application System Division
Center for Programming Science and Technology
Institute for Computer Sciences and Technology
U.S. Department of Commerce
National Bureau of Standards
Washington, DC 20234

January 1981

U.S. DEPARTMENT OF COMMERCE, Philip M. Klutznick, *Secretary*
Jordan J. Baruch, *Assistant Secretary for Productivity, Technology, and Innovation*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

M7:
A General Pattern Matching Facility
Users Manual

A. R. Marriott
G. H. Skillman
S. B. Salazar
W. T. Hardgrave

November 1980

Version 3.0

Application Systems Division
Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C.

M7 is a pattern matching and replacement facility developed as a UNIX tool for translating and reformatting queries, languages, and data. M7 operates by first preprocessing a set of user defined macros, then using these macros to match and replace the text in an input string. The enabling of the rescan option directs M7 to match and rematch the macro patterns against the input string until all possible replacements have been made. Three constructions--tags, stacks, and counters--allow communication between different macros and different input strings, to permit such functions as line numbering, labeling, and argument passing. This paper includes a tutorial which shows how to construct a set of macros to solve a typical problem using, as an example, a piece of a FORTRAN to C translator.

FEDERAL INFORMATION PROCESSING STANDARD SOFTWARE SUMMARY

01. Summary date (Yr. Mo. Day): 80 12 03

02. Summary prepared by (Name and Phone): Sandra B. Salazar X3491

03. Summary action: New [X] Replacement [ ] Deletion [ ]
Previous Internal Software ID:

04. Software date (Yr. Mo. Day): 80 09 11

05. Software title: M7: A String Pattern Matching Facility

06. Short title: M7

07. Internal Software ID:

08. Software type:
- [ ] Automated Data System
- [X] Computer Program
- [ ] Subroutine/Module

09. Processing mode:
- [ ] Interactive
- [ ] Batch
- [X] Combination

10. Application area

General		Specific
[ ] Computer Systems Support/Utility	[ ] Management/Business	Pattern Matcher
[ ] Scientific/Engineering	[ ] Process Control	
[X] Bibliographic/Textual	[ ] Other	

11. Submitting organization and address:

Application Systems Division, 642
National Bureau of Standards
Department of Commerce
Washington, DC 20234

12. Technical contact(s) and phone:

S. B. Salazar 921-3491
W. T. Hardgrave

13. Narrative:

M7 is a pattern matching and replacement facility developed as a UNIX tool for translating and reformatting queries, languages, and data. M7 operates by first preprocessing a set of user defined macros, then using these macros to match and replace the text in an input string. The enabling of the rescan option directs M7 to match and rematch the macro patterns against the input string until all possible replacements have been made. Three constructions -- tags, stacks, and counters -- allow communication between different macros and different input strings, to permit such functions as line numbering, labeling, and argument passing.

14. Keywords:

macroprocessor; pattern matching; query processor; software tool; text processor; translation; UNIX.

15. Computer manuf'r and model	16. Computer operating system	17. Programing language(s)	18. Number of source program statements
PDP-11	UNIX	C	3K
19. Computer memory requirements	**20. Tape drives**	**21. Disk/Drum units**	**22. Terminals**
64K	NA	1	1

23. Other operational requirements:

24. Software availability:
- [X] Available
- [ ] Limited
- [ ] In-house only

25. Documentation availability:
- [X] Available
- [ ] Inadequate
- [ ] In-house only

26. FOR SUBMITTING ORGANIZATION USE

INSTRUCTIONS

01. **Summary Date.** Enter date summary prepared. Use Year, Month, Day format: YYMMDD.

02. **Summary Prepared By.** Enter name and phone number (including area code) of individual who prepared this summary.

03. **Summary Action.** Mark the appropriate box for new summary, **replacement** summary or **deletion** of summary. If this software summary is a replacement, enter under "Previous Internal Software ID" the internal software identification as reported in item 07 of the original summary, and enter the new internal software identification in item 07 of this form; complete all other items as for a new summary. If a software summary is to be deleted, enter under "Previous Internal Software ID" the internal software identification as reported in item 07 of the original summary; complete only items 01, 02, 03 and 11 on this form.

04. **Software Date.** Enter date software was completed or last updated. Use Year, Month, Day format: YYMMDD.

05. **Software Title.** Make title as descriptive as possible.

06. **Short Title.** (Optional) Enter commonly used abbreviation or acronym which identifies the software.

07. **Internal Software ID.** Enter a unique identification number or code.

08. **Software Type.** Mark the appropriate box for an **Automated Data System** (set of computer programs), **Computer Program**, or Subroutine/Module, whichever best describes the software.

09. **Processing Mode.** Mark the appropriate box for an **Interactive, Batch**, or **Combination** mode, whichever best describes the software.

10. Application Area.
    General: Mark the appropriate box which best describes the general area of application from among:
    Computer Systems Support/Utility — **Process Control**
    Management/Business — **Bibliographic/Textual**
    Scientific/Engineering — **Other**
    Specific: Specify the sub-area of application; e.g.: "COBOL optimizer" if the general area is "Computer Systems Support/Utility"; "Payroll" if the general area is "Management/Business"; etc. Elaborate here if the general area is "Other."

11. **Submitting Organization and Address.** Identify the organization responsible for the software as completely as possible, to the Branch or Division level, but including Agency, Department (Bureau/Administration), Service, Corporation, Commission, or Council. Fill in complete mailing address, including mail code, street address, city, state, and ZIP code.

12. **Technical Contact(s) and Phone:** Enter person(s) or office(s) to be contacted for technical information on subject matter and/or operational aspects of software. Include telephone area code. Provide organization name and mailing address, if different from that in Item 11.

13. **Narrative.** Describe concisely the problem addressed and methods of solution. Include significant factors such as special operating system modifications, security concerns, relationships to other software, input and output media, virtual memory requirements, and unique hardware features. Cite references, if appropriate.

14. **Keywords.** List significant words or phrases which reflect the functions, applications and features of the software. Separate entries with semicolons.

15. **Computer Manufacturer and Model.** Identify mainframe computer(s) on which software is operational.

16. **Computer Operating System.** Enter name, number, and release under which software is operating. Identify enhancements in the Narrative (item 13).

17. **Programing Language(s).** Identify the language(s) in which the software is written, including version; e.g., ANSI COBOL, FORTRAN V, SIMSCRIPT II.5, SLEUTH II.

18. **Number of Source Program Statements.** Include statements in this software, separate macros, called subroutines, etc.

19. **Computer Memory Requirements.** Enter minimum internal memory necessary to execute software, exclusive of memory required for the operating system. Specify words, bytes, characters, etc., and number of bits per unit. Identify virtual memory requirements in the Narrative (item 13).

20. **Tape Drives.** Identify number needed to operate software. Specify, if critical, manufacturer, model, tracks, recording density, etc.

21. **Disk/Drum Units.** Identify number and size (in same units as "Memory"—item 19) needed to operate software. Specify, if critical, manufacturer, model, etc.

22. **Terminals.** Identify number of terminals required. Specify, if critical, type, speed, character set, screen/line size, etc.

23 **Other Operational Requirements.** Identify peripheral devices, support software, or related equipment not indicated above, e.g., optical character devices, facsimile, computer-output microfilm, graphic plotters.

24. **Software Availability.** Mark the appropriate box which best describes the software availability from among: **Available to the Public, Limited Availability** (e.g.: for government use only), and **For In-house Use Only.** If the software is "Available", include a mail or phone contact point, as well as the price and form in which the software is available, if possible.

25. **Documentation Availability.** Mark the appropriate box which best describes the documentation availability from among: **Available to the Public, Inadequate for Distribution,** and **For In-house Use Only.** If documentation is "Available", include a mail or phone contact point, as well as the price and form in which the documentation is available, if possible. If documentation is presently "Inadequate", show the expected availability date.

26. **For Submitting Organization Use.** This area is provided for the use of the organization submitting this summary. It may contain any information deemed useful for internal operation.

GPO : 1974 O - 554-655

TABLE OF CONTENTS

1. INTRODUCTION

M7 is a general pattern matching filter designed and implemented at the National Bureau of Standards (NBS). It is a useful tool for translating or reformatting queries, languages, and data. M7 is particularly useful and cost-effective for one-time translations from one format to another or in a research environment to demonstrate the feasibility of a transformation.

M7 repetitively matches and replaces the text on an input string under the control of a set of user defined macros. The process consists of two stages: (a) preprocessing the macro file, and (b) matching and replacing the input string. In the first phase, macros are read from the user's file, preprocessed, and stored on a second file. In stage two, the patterns are compared, in a line-by-line manner, against input strings read from standard input. Matching and replacing continues until all of the patterns fail to match the input. The final version of the altered input string is sent to standard output. Figure 1.1 shows the M7 information flow.

Complete examples of the use of M7--macros, input strings, output text--are available on-line from the authors.

1.1 BACKGROUND

M7 is written entirely in the programming language C and consists of more than 40 modular subroutines and functions. Several routines are C versions of programs contained in Software Tools [KERN76] which have been modified to support the more powerful features of M7. M7 is an extension of M6 [HALL71] and MORTRAN [COOK73] and has also been influenced by ML/1 [BROW74,COLE76] and STAGE2 [WAIT70]. For more information about the internals of M7 see the M7 Software Internals Manual [SKIL80].

1.2 DOCUMENT OVERVIEW

This document presents the information necessary for using M7. Section 2 describes the macro file. Sections 3 and 4 discuss the pattern and the replacement definition, emphasizing the characters and constructions that have special meanings. The use of stacks to save text for later matching and replacement is described in section 5. Section 6 deals with counters which can be used for tasks such as

line numbering. Section 7 discusses how M7 uses the macro file to repetitively match the text on an input string. Section 8 describes how to generate new macros from other macros and section 9 lists the calling options for M7. A detailed example which shows how to create and arrange macros to solve a specific problem is provided in section 10. Error messages and constraints are the topics of the closing sections.

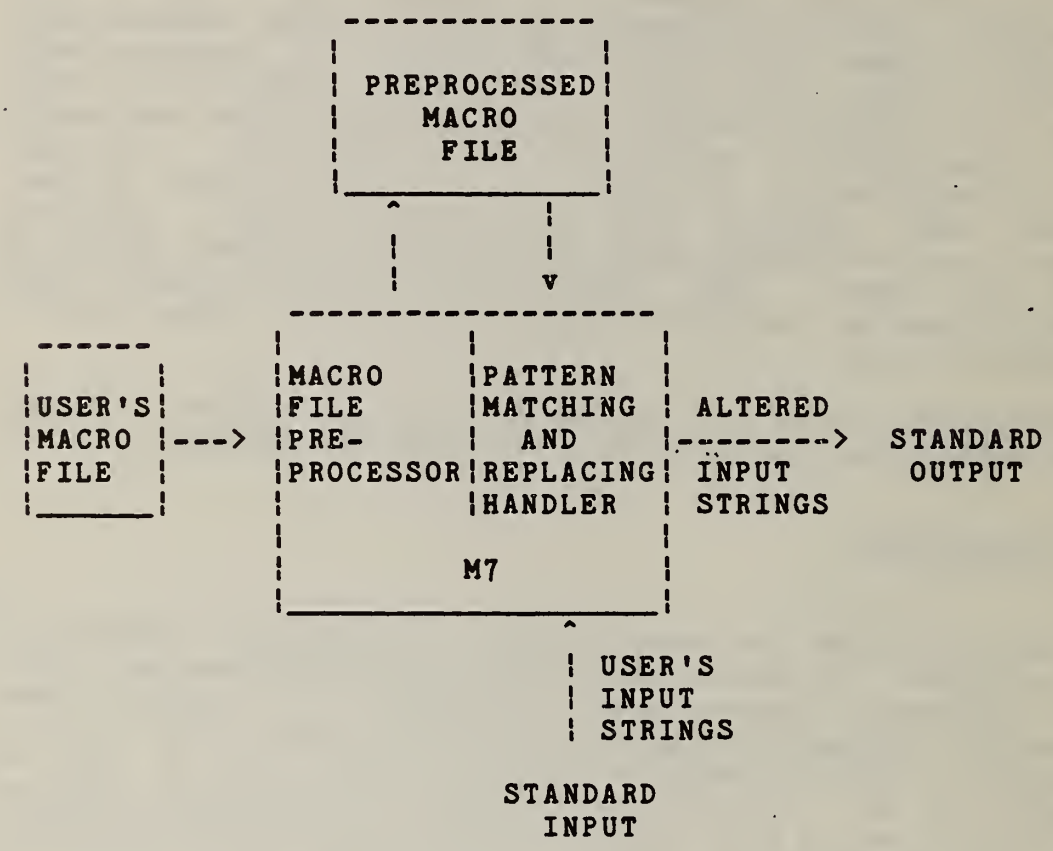

Figure 1.1 - M7 Information Flow

2. MACRO FILE

M7 is called from the UNIX shell level in the following manner:

M7 <options> <macro file>

The macro file contains the pattern matching and replacement information which M7 evaluates when processing input text. It consists of macro definitions which have two main parts: a pattern which is matched against an input string, and a replacement definition which is substituted for the matched substring. The options specify execution alternatives.

This section will explain how to set up the macro file.

2.1 MACRO DEFINITION

The macro file consists of macro definitions. The basic form of a macro definition is:

'<pattern>' <replacement symbol> '<replacement definition>' <stack/counter commands> ;

The symbols surrounded by angle brackets (<>) are meta-symbols; however, single quotation marks must surround both the pattern and the replacement definition. The macro definition must terminate with a semi-colon. The replacement symbol can be either "=" or "<" as discussed in section 7.

Stack and counter commands are usually placed after the replacement definition. These constructions are identified by a leading "#" or "&".

2.2 FILE FORMAT

The use of delimiting characters for the pattern, replacement definition, and the entire macro definition provides flexibility in formatting the macro file. The user may insert spaces and comments freely to make the file more readable.

Arbitrary spacing is allowed. For example:

```
'A'     =       'B';
'A'='B'         ;
'A'=    'B';
```

all function identically.

All characters outside of the pattern and the replacement definition, other than the replacement symbol and the stack and counter commands, are ignored. Thus, no special delimiter is needed for most comments. An example of a commented macro file is:

```
'boy'   =   'girl';   this macro changes boy to girl
' '     =   ' ';      this macro shrinks spacing
'a'     =   '';       this macro deletes the character a
(end).
```

Any text surrounded by slashes is ignored. This feature allows comments to be inserted within a pattern or replacement definition. It also permits arbitrary splitting of a macro definition across line boundaries. For example:

```
'FIND THIS TEXT' = 'AND REPLACE IT /
                  / WITH THIS TEXT';
```

A macro definition may also be continued by placing the pattern and its replacement definition on separate lines. An example of this is:

```
'Replace this very long line'   =
'with this very long line';
```

More than one macro definition can be placed on a line. For example:

```
'A'='B';
'C'='D';
```

can be written as:

```
'A'='B'; 'C'='D';
(end).
```

3. PATTERNS

The pattern matching facility is the heart of M7. Sections 3.2 and 3.3 will discuss the important features: the special control characters, tags, stacks, and counters. The algorithm for matching a pattern with an input string will be presented in section 3.1.

3.1 GENERAL PATTERN MATCHING

The algorithm for pattern matching is as follows:

1. Get the first character of the input string
   and set N := 1.
2. Get the first character of the pattern.
3. Do the current pair of characters match?
   If not, go to step 5.
4. Get next pair of characters.
   If end of pattern, return MATCH.
   Otherwise, go to step 3.
5. Backup to beginning of input string + N.
   If N = length(input string), return FAIL.
   Otherwise, set N := N+1;
   get Nth character of input string;
   go to step 2.

3.2 SPECIAL CHARACTERS

Table 3.1 lists the characters which have special meaning in patterns. These characters and their associated concepts will be described below.

Delimiting characters

A '$' matches the null character at the end of a line. For example, the pattern:

```
The back$
```

will match:

```
The back
```

but not:

```
The back end
```

(end).

A '^' matches the null character at the beginning of a line. For example, the pattern:

```
^front part
```

will match:

```
front part
```

but not:

```
the front part
```

(end).

CHARACTER	FUNCTION OF THE CHARACTER
$	Matches the null character at the end of the line.
^	Matches the null character at the beginning of the line.
*	Matches one or more characters of the preceding type of character. (Closure 1)
*-	Matches zero or more characters of the preceding type of character. (Closure 2)
[]	Matches any of the characters listed in the brackets. (Character class)
-	Indicates a range of characters in a character class.
~	Matches anything not listed in a character class. (Complement)
?n	Matches a character that is in special character class n. 1 <= n <= 6
&	Refers to an entry on one of 26 stacks.
#	Refers to a counter or its increment.
{}	Tags or refers to a portion of input text.
/	Skips to the next slash. (Line continuation)
\	Escapes any of the special characters.

Table 3.1 - TABLE OF SPECIAL CHARACTERS

Character class

The construction "[c1c2c3...cn]", termed a character class, tells M7 to match one of the characters specified between the brackets (i.e. c1, or, c2, or, ... cn). For example, the pattern:

```
[abcdef]
```

will match one of the first six letters of the alphabet.

A range of letters or digits can be specified using a dash. For example, the pattern:

```
[a-c]
```

will match 'a', 'b', or 'c'. Also, the pattern:

```
[0-3]
```

will match '0', '1', '2', or '3'.

The use of '~' as the first character in the character class reverses the meaning of the construction, so that a match will occur only if the input character is not found between the brackets (complement). For example, the pattern:

```
[~a-z]
```

would match any character that is not a lower case letter.

Several frequently used character classes can be abbreviated by using the construction "?n", where n is a digit between one and six. These are:

```
?1 - matches any character.
?2 - matches any alpha-numeric character.
?3 - matches any alphabetic character.
?4 - matches any upper case letter.
?5 - matches any lower case letter.
?6 - matches any digit.
```

Escape character

A backslash '\' escapes the special meaning of the character which follows it. For example, the pattern:

```
\*
```

matches the character '*'. The escape character can be escaped by typing "\\". Two alphabetic characters have

special meaning when they are escaped. "\n" matches a line feed and "\t" matches a tab.

The following characters lose their special meanings under these conditions:

Character	Condition
-	At the end of a character class construction or outside of a character class construction.
~	Not at the beginning of a character class construction or outside of a character class construction.
?	Not followed by a digit between 1 and 6.
Any special character other than "~" or "-"	Within a character class.

Closure

Closure is an important concept in M7. The closure character "*" matches one or more additional characters that meet the same specifications required of the preceding character. For instance, the pattern:

```
[ab]
```

will match a character if it is an "a" or "b". Thus, the pattern:

```
[ab]*
```

will match a string of "a"'s and "b"'s. Also, the pattern:

```
a*
```

will match a string of "a"'s, the pattern:

```
[~a-zA-Z]*
```

will match a string of non-alphabetic characters and the pattern:

```
?1*
```

will match any text.

The closure character repeats the previous pattern construction and therefore a pattern cannot begin with "*", "{*}", or "^*", since there is no previous pattern with which to check succeeding characters in the input. The closure construction may only follow stack and counter calls that refer to the input string (namely, type 1 - refer to

chapters 5 and 6). The construction "***" is meaningless since the second closure has no pattern preceding it.

Closure algorithm

M7's closure feature uses the following algorithm:

1. Match the longest string possible.
2. Does the rest of the pattern match?
   If yes, indicate success and terminate.
3. Did the previous character match this closure?
   If yes, back up one character and go to step 2.
4. Was there a previous closure?
   If yes, back up one character from the last character matched in the previous closure and go to step 2.
5. Indicate failure and stop.

An illustration of multiple closure patterns, which are handled by the fourth step in the algorithm, is the pattern:

```
?1*[0-9]*
```

with the input text:

```
Mozart's 38th symphony
```

M7 will decrease the number of characters matched by the first closure until the second closure is able to match the '8' upon which the pattern succeeds.

Zero closure

The zero closure "*-" matches zero or more characters that meet the same specifications required of the preceding character. It can be used in the same manner as the regular closure.

Example

Here are a few additional sample patterns and some of the input texts that would match:

PATTERN		TEXT
?4A*-B	would match	TAB
	or	JB
^[1-3][ABC]?3$	would match	1BE
	or	3AT
??3*	would match	?singleword
	but not	??333

3.3 TAGS, STACKS AND COUNTERS

Tags, stacks and counters are three features that distinguish M7 from other pattern matching and replacement programs. These structures allow communication between:

a. the pattern and replacement definitions of a macro,
b. independent macros, and
c. different input strings.

The usage and form of a tag, stack, or counter construction depends on whether it appears on the pattern or the replacement side of the macro definition. Basically, on the pattern side, the current value of a tag, stack, or counter is matched against the input string, whereas on the replacement side, the current value of the tag, stack, or counter is used to alter the input string. The applicability of these constructions to the pattern will be discussed in this section, while their relevance to the replacement definition will be presented in section 4. Sections 5 and 6 will discuss stacks and counters in detail.

Tags

The construction "{<pattern>}" tells M7 to remember the text which matches the pattern between the braces for use in later processing. The occurrences of these "tags" in a pattern are numbered; the first is numbered 1 and so forth with up to 99 tags per macro. For example, if the pattern is:

```
h{[aeiou]*}d
```

and the input text is:

```
head
```

then M7 will remember "ea". Tagged text can be used in the

replacement definition or stored on a stack.

Tags can be nested in the pattern. For instance, the pattern:

```
{ab{cd}e{f{ghi}}} {j}
```

will remember "abcdefghi" as tag #1, "cd" as tag #2, "fghi" as tag #3, "ghi" as tag #4, and "j" as tag #5. That is, the left brace determines the ordering.

Stacks

The construction "&(i)", where "i" is any lower case letter, causes M7 to match what is currently being referenced in the stack identified by "i". For example, if stack "a" contained the text "tony" then the pattern:

```
He is &(a)
```

would match:

```
He is tony
```

and the pattern:

```
the world needs more &(a)*s
```

would match:

```
the world needs more tonytonytonytonys
(end).
```

Counters

The construction "#(i)", where "i" is any lower case letter, matches the current value of the counter identified by "i". For example, if counter "b" were "20" then the pattern:

```
go to #(b)*
```

would match:

```
go to 2020
(end).
```

4. REPLACEMENT DEFINITION

When a portion of the input string matches a pattern, it is replaced by text as specified by the replacement definition. The replacement definition can consist of any sequence of characters. As in pattern matching, certain characters have special meaning.

Tags

The construction "{n}" is replaced by the text matched by tag number n where n is from 1 to 99. This tag number refers to an occurrence of a tag in the corresponding pattern (see section 3.3). The first tag would be tag number 1, the second tag would be tag number 2 and so forth. The construction is useful for such tasks as passing arguments from the matched text. For example, suppose there is a pattern:

```
ADD,{[A-Z]*},{[0-9]*},{[0-9]*}
```

with replacement definition:

```
{1}={2}+{3}
```

Then, the input string:

```
ADD,A,24,100:
```

would be replaced by:

```
A=24+100
```

(end).

Stacks

The construction "&(i)" is replaced by the text currently referenced in stack "i". For example if stack "a" contained "car", then the replacement definition:

```
my &(a)
```

would cause M7 to replace some matched string with:

```
my car
```

A detailed discussion of stacks can be found in section 5.

Counters

The construction "#(i)" is replaced by the current value of the counter "i". For example, if counter "b" were 100 then the replacement definition:

```
go to #(b)
```

would cause M7 to replace some matched string with:

```
go to 100
```

Refer to section 6 for a detailed discussion of counter constructions.

5. STACKS

M7 supports 26 user stacks. Each stack is identified by a lower case letter, called the stack identifier, and has its own stack pointer. In the current version, a stack has at most twenty entries, each of which may have fifty characters. The purpose of the stacks is to save text for later use in the matching and replacement parts of other macro definitions. The stacks may also be used as simple variables.

Stack call constructions

There are four basic stack call constructions. All of these may appear in the replacement definition. Types 2, 3, and 4 may be used in the stack command. Only type 1 may appear in the pattern. The stack call constructions are:

1. "&(i)" is replaced by what is currently being pointed to in the stack identified by 'i'.

2. "&(n,i)" puts the text matched by tag number n onto stack 'i' where n is from 1 to 99.

3. "&(i=n)" sets the stack pointer for stack 'i' to n where 1 <= n <= 20.

4. "&(i:<text>:)" puts the text onto stack 'i'. (<text> is a meta-symbol.)

Spaces are ignored in stack calls.

Stack pointer

An optional feature of types 1, 2, and 4 stack calls are the two stack operators, "+" and "-", which respectively increment and decrement the stack pointer by 1. As in the language C, placement of the operators before or after the stack identifier indicates whether the operation is performed before or after the stack is accessed. For example:

```
&(+e)
```

would first increment the stack pointer and then be replaced by what is currently being pointed to in the stack "e". The stack operators should not be used on stack calls which appear in the pattern. If the stack operators are not used, a stack can be viewed as a simple variable.

The stack pointer is always initialized to point at position 1 and cannot be decremented below this position. If the stack pointer points at position 1 and a decrement is indicated, the stack pointer will continue to point at position 1. Before text is placed onto a stack position, that position contains the null string.

6. COUNTERS

M7 allows the user to have 26 general purpose counters each of which has its own counter increment. The counters are physically stored as integers, but they are used in the pattern and the replacement definition in their character string form. Conversion is done automatically. Each counter is identified by a lower case letter called the counter identifier. These counters are useful for tasks such as manipulating line numbers and counting how many times a pattern is matched.

Counter call constructions

There are three basic counter call constructions. While all types may be used in the replacement definition, only type 1 may appear in the pattern and only types 2 and 3 are applicable to the counter command. The counter call constructions are:

1. "#(i)" is replaced in the string by the current value of the counter identified by "i".

2. "#(i=n)" sets the counter to n where n can be any integer greater than 0.

3. "#(i,n)" sets counter i's increment to n where n is a positive integer.

Spaces are ignored in counter calls.

Counter increment

The two counter operators, "+" and "-", are similar to the stack operators except that stack pointers are incremented or decremented by 1 whereas a counter is changed by the value of its increment. These operators can be placed before or after the counter identifier to indicate that the counter should be incremented or decremented before or after the particular function is performed. For example:

```
#(y+)
```

would be replaced by the current value of the counter 'y' after which the counter would be incremented.

Counters and increments are always initialized to 1 and cannot be set to a value less than 1. As with stacks, the operators should not be used in the pattern part of a macro.

Example of Tags, Stacks, and Counters

For an example of how tags and stacks should be used, consider the macro:

```
'avg({[0-9]*},{[0-9]*})' = / notice the
        comment / '({1}+{2})\/2'&(1,a)&(2,b);
```

This would match the input:

```
avg(26,42)
```

and replace it with:

```
(26+42)/2
```

and then save the arguments "26" and "42" on stacks "a" and "b", respectively. An example of the use of a counter would be the macro:

```
'reset a' = 'done'#(a=1);
```

This would match the input:

```
reset a
```

and replace it with:

```
done
```

and set counter "a" to "1".

7. RESCAN FEATURE

M7 will repetitively match and rematch the text on an input string according to the macro definitions until no more matches can be found. The algorithm for the matching and replacement of input strings is as follows:

1. Read the next input string from the standard input.
2. If it is the end of the file then stop.
3. Get the first pattern.
4. Replace all occurrences of the pattern in the input with the replacement definition.
5. Did the current pattern occur in the input at least once?
   If yes, go to step 3.
6. Is there another pattern?
   If yes, get the next pattern and go to step 4.
7. Write out the new line and go to step 1.

The "first pattern" refers to the most recent macro definition entered onto the macro file. Thus the first pattern attempted to be matched would be taken from the last macro definition in the macro file. The next pattern attempted to be matched would be from the next to the last entry and so on. Since the user can emit macros from other macros, as discussed in section 8, these later generated macros will always be scanned first.

The user should be careful not to cause M7 to go into an infinite matching loop. A very simple example of this would be the macro:

```
'int' = 'integer';
```

The pattern "int" would match the input "print" and replace it with "printeger". M7 will again use this macro to continuously match "int" and replace it with "integer". The user should use the trace feature (see section 9.0) to make sure this type of replacement does not occur.

The rescan feature can be turned off for a particular macro by using "<" as the replacement symbol instead of "=". M7 will attempt to change all occurrences of the pattern in the input string to what is found in the replacement definition. If successful, the pattern will not be used again until the next input string is read in. Thus, this construction can be used to avoid infinite matching loops.

For example, the macro definition:

```
'ab'<'abc';
```

and the input:

```
ababab
```

will result in:

```
abcabcabc
```

(end).

8. CONTROL COMMANDS

Any input string which begins with the character "%" is considered a control command. The line will not be output nor will an attempt be made to match any other patterns against it. The legal commands and their uses are:

%MACRO - generates a new macro definition
%TRACE - enables/disables the trace (-t) option
%NFLAG - enables/disables the print (-n) option

The format for the "%MACRO" command is:

```
%MACRO '<macro definition>' ;
```

This control command construction enables generation of macros from other macros. The new macros are placed at the end of the macro file, so that they will be scanned first. For example, consider the macro:

```
'#define,{[a-z]*},{[a-z]*}' = /
 / '%MACRO \'{1}\'=\'{2}\'\;';
```

The pattern of this macro would match the input "#define,cat,dog" and replace it with "%MACRO 'cat'='dog';". The modified input string would be evaluated as a control command structure and placed onto the preprocessed macro file. The new macro would be the first macro to be scanned. If the next input were "cat" it would be replaced by "dog".

Program flags can be set by typing:

```
%<flagname> 1(or 0)
```

where flagname can be "TRACE" or "NFLAG". The numbers 1 and 0 stand for ON and OFF respectively. For example, the input:

```
%TRACE 1
```

will turn the trace option (the "t" option) on and the input:

```
%NFLAG 1
```

will turn the n option on. These options are discussed below. This construction can also be generated from a macro as described above.

A command is not successful if an error occurs in the macro of a macro generation command or if a symbol other than '0' or '1' is the eighth character of the '%TRACE' or '%NFLAG' command. If an illegal macro is given, the same error messages are displayed as when M7 is preprocessing the input macro file; if an illegal value for a program flag is given, an error message is displayed but execution does <u>not</u> terminate.

9. EXECUTION OPTIONS

M7 has several calling options any one or all of which may be specified using the standard UNIX calling procedure:

```
M7 [-p] [-t]
[-f "<preprocessed file>" "<macro count>"]
[-a "<macro definition>"] <macro file>
```

The following is a list of the options and their functions.

1. The -t option will print a trace of the pattern matching and replacement on the standard output. This is very useful for "debugging" macro files. The trace is of the form

   ```
   oldline:<text before replacement>
   macro #:n
   newline:<text after replacement>
   ```

   where n is the macro number. (Numbers start with the first entry in the preprocessed macro file.)

2. The -n option prints only the input strings which were matched by at least one macro definition.

3. The -p option provides a prompt for initial input. After M7 has preprocessed the macro file, "ready" will be printed to inform the user that he can start typing in input strings.

4. The -f option specifies a file which contains macros that are already preprocessed. With this feature, the user need not wait for M7 to repreprocess a commonly used macro file. The format for this option is:

   ```
   -f "preprocessed file" "macro count"
   ```

   where the preprocessed file is the file of preprocessed macros and macro count is a count of the number of macros in the file. A typical example would be:

   ```
   M7 -f "M7_WKS.tmp" "23"
   ```

   The "f" option must be placed before any occurrence of the "a" option and before the names of any files which contain non-preprocessed macros. If the "f" option is used, M7 will work with the specified file

instead of creating the new file "M7_WKS.tmp". The user should refer to section 12 for information pertaining to the limitations of the 'f' option.

5. The -a option takes the following character string, surrounded by double quotes, as a macro definition. This option may be repeated several times to put several macros on the file. These macros will be the first macros preprocessed and consequently will be used after any macros in an unprocessed macro file.

6. More than one macro file may be specified in the argument list. The effect will be as though the files were concatenated into a single file, with the macros in the latter files at the end.

10. USING M7

This section demonstrates a practical application of M7. A systematic procedure for source translations using pattern matching macros will be presented along with tips and cautions on how to set up a macro file.

The first step in using M7 is to define what the input and the output should look like. If the input and output are well defined, fewer problems will be encountered while writing the macros. Restrictions will usually have to be put on the initial specifications because of implementation limitations. The user should be aware of all the possible combinations of input and be willing to change the specifications as necessary. The example below translates from a FORTRAN-like DO statement to a C-like "for" statement.

FROM:

```
      do <label> <index>=<init value>,<final value>,<inc>
      <stmt1>
      <stmt2>
      .
      .
      .
      <stmtn>
<label> continue
```

TO:

```
      for(<index>=<init val>;<index><=<final value>;
                             <index>=<index>+<inc>){
      <stmt1>;
      <stmt2>;
      .
      .
      .
      <stmtn>;
      }
```

A pictorial representation of this type is one way to specify input and output. Another way is to set up a table of possible inputs and the corresponding outputs. Do not forget details. For example, the specifications of this example failed to show that the increment need not be specified in FORTRAN; it defaults to 1. Details such as these should be included in the input and output specifications.

The next step would be to design a pattern matching algorithm for the translation. This can be written out as a step by step word description of the pattern matching and replacement. For this example, the algorithm is:

1. Shrink the spacing (i.e. replace tabs and double spacing with a single space)

2. Match a FORTRAN "do" string and replace it with a C programming language "for" string and store the label on a stack.

3. Match a statement which does not have a semicolon or a brace at the end and replace it with a statement with a semicolon at the end.

4. Match the label stack at the beginning of a continue statement and replace it with a right brace.

5. Strip off the label field.

The idea of shrinking spaces is an important utility and is used often.

The next step after the algorithm is written out is to create a set of macros which will perform each step of the algorithm. As they are written, each set should be tested separately for correct output.

Precedence is important at this point. The ordering of the macros in the macro file has a significant influence on the output because of the rescanning algorithm. The macros of higher precedence or ones that other macros depend on should be placed closer to the end of the macro file so that they are applied first. Space shrinking macros are usually placed near the end of the macro file. The trace feature can be used to see how the steps in the translation algorithm interact with each other. The following set of macros perform the translation which was described previously:

```
1. '{[~]{\;]}$' = '{1}\;';                                    step 3
2. 'do {?6*} {?2*}={?6*},{?6*}$'=                             step 2
   'for({2}={3}\;{2}<={4}\;{2}++)\{',&(1,a);
3. 'do {?6*} {?2*}=/                                          step 2
   /{?6*},{?6*},{?6*}' =
   'for({2}={3}\;{2}<={4}\;{2}={2}+{5})\{',&(1,a);
4. '^[ 0-9]*' = '';                                           step 5
5. '^&(a) *continue' = '\}';                                  step 4
6. '^ &(a) *continue' = '\}';                                 step 4
7. '  ' = ' ';                                               step 1
8. '\t' = ' ';                                                step 1
```

The step numbers correspond to the step numbers of the word description of the translation algorithm given above.

The first macro places the semicolon at the end of a statement. This macro is at the beginning of the file because it is the last macro to be matched. Otherwise, M7 would put a semicolon at the end of the "do" statement before translating it which is not what is desired. Note that all semicolons which appear in the macro file which do not terminate a macro definition must be escaped.

The macros numbered 2 and 3 translate the FORTRAN "do" statement into a C programming language "for" statement. The ordering of these two macros in the macro file is significant. The third macro matches the optional increment specification. The second macro will also match this structure. If the second macro were placed in front of the third macro (i.e. further down in the file) the output would be incorrect.

The fourth macro removes numbers and spaces which occur at the beginning of a line (i.e. removes the line numbers).

The patterns of the fifth and the sixth macros match the end of the "do" loop (i.e. the label indicated on the "do" statement, which was stored on a stack, is matched to the beginning of a continue statement). Note, that this pattern is matched before the pattern of the macro that removes the line numbers.

The seventh and eighth macros shrink the spacing; double spaces and tabs are replaced with single spaces. This is done so that the patterns of macros 2 and 3 will match arbitrary spacing of the "do" statement.

The next step after the macros have been written and checked for precedence is to test them on a wide range of input. Usually errors or limitations in the original algorithm can be found at this step. For example, after testing this set of macros one would find that nested "do" loops would not work. This can be corrected by changing the macros so that the label indicated in the "do" statement is pushed onto the stack and then popped off when the corresponding "continue" statement is found.

Creating a set of macros to perform a desired translation is often as complex as writing a computer program; as with a computer program the more time spent on the input and output specifications and the matching algorithms the less time spent on trial and error writing of macros.

Consider the matching process when using stacks or counters. Macros should be written so their contents can be dumped. For example:

```
'dump a' = '&(a)';
```

would dump the contents of stack a. One should be careful about pointing to the right position in a stack or using the right value of a counter. Remember that the increment and the decrement operators have different meanings when placed before or after a stack identifier.

It is very easy to confuse M7. The user should check the following things when problems occur:

1. that single quotes occur around the pattern and replacement definition.

2. that a semicolon occurs at the end of every macro definition.

3. that special characters which are to be used as text are escaped.

4. that semicolons which do not occur at the end of a macro definition are escaped.

11. FATAL ERROR MESSAGES

All errors detected by M7 (except illegal values for the t and n options) result in termination of execution. This is done because execution after fatal errors is meaningless. The error messages that are generated are of the form:

<routine name>:<reason for termination>
The error occurred on macro # <macro number>

where the routine name is the name of the M7 subroutine where the error was detected. (See the M7 Software Internals Manual)

The error messages which occur during the preprocessing of the macro file usually occur because of things like missing quotes or unescaped special characters. The macro file should be thoroughly checked when an error occurs in preprocessing. The following is a list of the error messages which are generated during preprocessing and their possible causes:

1. "M7: cannot open pattern file" indicates that M7 could not open the specified macro file. The user should check the calling arguments used.

2. "M7: cannot open 'f' option file" indicates that M7 could not find the already existing preprocessed macro file.

3. "M7: illegal placement of "f" option" indicates the "f" option was used after the "a" option or after the name of a user macro file was given.

4. "PROCCALLS: error in macro definitions" indicates that M7 reached the terminating character ';' before it had finished processing the macro definition.

5. "MAKPAT: pattern terminated early" indicates that M7 found the end of string character, EOS, in the pattern.

6. "MAKPAT: unbalanced tag braces" indicates M7 found an unequal number of left and right braces in the pattern.

7. "MAKSUB: substitution text terminated early" indicates the end of string character, EOS, was found before the terminating semicolon.

8. "MAKSUB: preprocessed macro too large" indicates a macro was entered which when preprocessed, expanded beyond the 512 character limit.

9. "PROCCNTR: UNRECOGNIZEABLE CHAR" indicates that a symbol other than '=' or ',' was found in the counter call. Thus, M7 could not recognize the type of counter call it was.

10. "PROCSTCK: ILLEGAL CHARACTER" same as above except for stacks. A symbol other than ',', '=', or ':' was found.

11. "PROCCNTR: ILLEGAL USE OF ','" and "PROCCNTR: ILLEGAL USE OF '='" means that M7 found more than one of the special symbols of a stack call and therefore could not determine the type. Again, this is a syntax error.

12. "PROCSTCK: ILLEGAL USE OF ','", "PROCSTCK: ILLEGAL USE OF '='" and "PROCSTCK: ILLEGAL USE OF ':'" same as above except for stacks.

13. "ESC: END OF STRING ENCOUNTERED TOO SOON" The end of string character, EOS, was found before the delimiting quote or semicolon.

14. "GETLINE: input string too long" indicates an input string or an input macro was entered which was longer than 1054 characters.

The error messages which occur after the preprocessing of the macro definitions (i.e. after M7 prints "ready") usually occur because of internal confusion. This may be caused by errors in the macro file which were not detected during preprocessing. The following is a list of the error messages which are generated after preprocessing.

1. "OMATCH: illegal pattern construction" indicates that M7 expected to find one of the internally coded commands but found gibberish instead. This usually occurs when the user has an illegal construction such as "**" in the pattern.

2. "DOSTCK:error in stack call" indicates that M7 found an invalid stack call construction.

3. "DOCNTR:error in counter call" indicates that M7 found an invalid counter call construction.

12. CONSTRAINTS AND LIMITATIONS

M7 has some program limitations which may be changed in future versions of M7. The following is a list of the known limitations and constraints in M7:

1. M7 does not indicate when it is in an indefinite matching loop. If M7 does not seem to be responding with any output, use the trace feature (see section 9.0) to see what is happening.

2. The use of stack and counter constructions, other than type 1, can lead to peculiar results if placed in the pattern. This is because M7 executes such constructions as it is scanning the macro and the input string. M7 actually makes several attempts to match a pattern before being successful and with each new attempt all the stack and counter calls are executed again. If, during the course of trying to match a pattern, M7 scans the pattern ten different times, then the stack and counter calls in the pattern will be executed ten times. The stack and counter calls placed before a macro's pattern will only be executed the first time M7 attempts to find an occurrence of the pattern in a particular input string while calls placed after a pattern will only be executed if the pattern matches. This is why the use of incrementation in a call within or before a pattern will almost certainly have disastrous results. For this reason only type 1 stack and counter calls should be used (and without any incrementation) in the pattern of a macro.

3. One possible reason for wanting to place such constructions in the pattern despite this warning would be to use this powerful macro:

```
'{?2}&(1,a) &(a)*-' < /delete reoccurrences/ '{1}';
```

   This macro will tag alpha-numeric characters which might be delimited by a space. The text that is tagged is immediately placed on a stack so that other occurrences of the text in the same input string may be deleted.

4. A preprocessed macro entry is restricted to 512 characters. This number can be changed by updating the source code and recompiling the new version.

5. An input string or input macro is limited to 1054 characters which is the size of eight lines of printer paper. This allows the user to put seven lines of header on his macro file to improve the looks of the file.

6. A stack entry is limited to 50 characters and each stack has space allocated for 20 entries (i.e. the stack pointer can legally be set to point at entries at positions 1 to 20 on a stack). This can also be changed by recompiling the source code.

7. Semicolons used for any reason other than terminating a macro definition must be escaped. This also applies to double and single quotation marks which do not delimit comments or sections of a macro.

8. File headers should be restricted to about 800 characters. If commentary text is too large a memory fault will occur.

9. Care should be taken in using stacks and counters. Although M7 checks for extraneous symbols, it does not check for out of place letters, digits and plus and minus characters.

10. The limitation on the number of macros with the '<' feature has been set to 100 in this version of M7. However, the number can be changed by updating the source code and then recompiling the entire program.

11. The macros in the file given with the 'f' option are treated as though they all had "=" as their replacement symbol. If the user needs to turn off the rescan feature of any macros in his file, he must go through the preprocessing stage each time he executes M7.

12. The name of the 'f' option file is limited to ten characters.

REFERENCES

BROW74 Brown, Peter J., Macro Processors and Techniques for Portable Software, London, New York, Wiley, 1974.

COLE76 Cole, Alfred John, Macro Processors, Cambridge, New York: Cambridge University Press, 1976.

COOK73 Cook, A. James, "A User's Guide to CDC 7600/6600 MORTRAN," SLAC Computation Group, Stanford, California, 1973.

HALL71 Hall, Andrew D., "The M6 Macro Processor," Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1971.

KERN76 Kernighan, Brian W. and P. J. Plauger, Software Tools, Addison-Wesley Publishing Company, 1976.

SKIL80 Skillman, G. H., et al., "The M7 Internals Manual," National Bureau of Standards, Washington, D. C., in preparation.

WAIT70 Waite, W. M., "The Mobile Programming System: STAGE2", Communications of the ACM, 13, 7, 415-421.

NBS-114A (REV. 9-78)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBSIR 80-2182	2. Gov't. Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE

M7: A General Pattern Matching Facility Users Manual

5. Publication Date

January 1981

6. Performing Organization Code

7. AUTHOR(S)

A. R. Marriott, G. H. Skillman, S. B. Salazar, W. T. Hardgrave

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, DC 20234

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

13. Type of Report & Period Covered

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

☑ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

M7 is a general pattern matching and replacement facility. It is based on such macroprocessors as M6, MORTRAN, and STAGE 2, and has been implemented in the language C on UNIX. By incorporating such features as stacks, counters, and tags, M7 is particularly useful for translating or reformatting queries, languages and data.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Macroprocessor; pattern matching; query processor; software tool; text processor; translation; UNIX.

18. AVAILABILITY

☑ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Sup. of Doc., U.S. Government Printing Office, Washington, DC 20402, SD Stock No. SN003-003-

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

19. SECURITY CLASS (THIS REPORT)	21. NO. OF PRINTED PAGES
UNCLASSIFIED	38
20. SECURITY CLASS (THIS PAGE)	**22. Price**
UNCLASSIFIED	$6.00

USCOMM-DC