mse, a multi stream editor

version 2.02, 24 June 2007

by Vladimir Lidovski

Table of Contents

1 Introduction To MultiStream Editor

MSE is a multi stream editor. A multi stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). It in some ways is similar to another stream editors (such as SED or AWK) but it can process binary data as well as text.

1.1 Notes on This Manual

1.1.1 Purpose

This manual is basically designed to be a reference manual, although instructional information has been included on some of the more advanced features of the MSE program. The purpose of the manual is to fully describe the this program.

1.1.2 Prerequisites for Understanding This Manual

1. Familiarity with the computer to be used and a working knowledge of file system, including how to change directories and start programs from the console prompt.
2. Ability to use EMACS or some other word processor to produce unformatted text files.

   (See the manual for your word processing program if you are unsure of how to do this.)

1.1.3 Documentation Conventions

The following visual cues have been used in this documentation to help you interpret the information presented.

italic type
: Used for anything that you must type exactly as shown.

  Italic type is also used for MSE commands and their arguments, reference to a specific part of a MSE table, or words that are given special emphasis.

bold type
: Used for information you must provide. For example, in place of the word **filename**, type in the name of a file.

ALL CAPITALS
: Used for directory names, file names, acronyms, and command names.

`Ctrl+key`
: The plus sign between key names means that you hold down the first key and press the second key. For example, *`Ctrl+c`* means hold the `Ctrl` key down and press c.

The contents of files will be shown as mono-spaced type. The computer's response to what is typed will also appear as mono-spaced type.

1.2 How Can MultiStream Editor Help Me?

The MSE program is useful for finding all occurrences of specified characters, words, or phrases in a text file or series of text files, and making some type of change to this data in a consistent way. The change may be done in every occurrence found or only when certain conditions are met.

MSE is like the "search and replace" feature in a text editor, except much more powerful because it allows you to make changes which take context into consideration. Beyond the search and replace feature, MSE can also be used to count words in a text, insert or remove text, or reorder parts of a text.

2 Creating And Using A Change Table

2.1 Creating a Change Table

In order to use the MSE program, you must have a text file that describes the changes you want made. This file is called a change table. You may either use an existing change table, modify an existing table or create your own table. The table can be executed by running the MultiStream Editor program as described in Section 2.2 [Using a Change Table], page 2. A change table can be created or modified using EMACS or almost any other word processor. If you are using some program other than EMACS, be sure that you save your document as an unformatted text file. The filename extension ".mse" or ".cct" is commonly used when naming MSE tables.

The simplest change table instruction will have a *searched-for item (search string)* in a pair of quotes (double or single). This will be followed on the same line by a space, a right wedge, and another space. Next will be the desired *replacement (replacement string)*, again in quotes. The right wedge is an integral part of the MSE command and is not enclosed within quote marks. The right wedge separates the search side of the table entry from the replacement side.

For example, suppose you wanted to change all occurrences of "house" to "home". In a small text file, you would probably make the changes yourself in your word processor. However, making the changes to a large file, or a whole series of files, could take a longtime, and typing errors might occur in the process.

The following simple MSE table could be used to accomplish this change quickly and accurately:

```
"house" > "home"
```

Input:

```
Our house is a very fine house.  We like our house.
```

Output:

```
Our home is a very fine home.  We like our home.
```

See Section 3.1 [Form of Changes], page 3 for a more detailed description of the format of a change table.

As you write your change table, it is very important to remember how the MSE program works: MSE "reads" your text file *one character at a time*. As the program reads a character, it tries to match it to a search string on the left side of the change table. If it matches an entry the program obeys the commands on the right side of the table and the replacement text is sent to the output. The piece of input that matched does not go to the output. Remember that MSE is a search and replace program. Once it finds what it is searching for, it replaces it with something else. If a character of text doesn't match any search string, it goes straight to the output and the MSE program reads the next character.

2.2 Using a Change Table

This section assumes that you have found the file MSE on your diskettes, and that it has been copied into your current directory or into a directory that has been included in your path.

When you run MSE, it requires from you three filenames: the changes file, the output file, and the input file, and optional log file. The changes file contains the instructions that tell MSE what to change. The output file is the file MSE will create as it applies the changes to the input file. The input file contains the text you want changed.

The MSE program doesn't actually change the input text file, it creates a new text file like your original input file, except the changes specified in the change table have been made to it. So, in the end you will have a "before" and "after" version of your file.

If you want multiple files combined into one output file, enter several input files. Any number of files can be combined in this manner.

If for any reason you need to stop the program after it has started, press `Ctrl+c`. If you were outputting to a printer, a few more lines may continue to print because the printer holds some of the information internally before printing it.

(See Section 3.5 [I/O Options], page 19 for a description of other I/O options available).

3 Changes Table Description

3.1 Form of Changes

A change file must be created before MSE is run. A change file is a text file which consists of one or more change entries. All change entries are of the form:

```
search  >  replacement
```

The search must fit on a single line. The right wedge (>) must be on the same line as the search. The replacement may be any number of lines. Blank lines are allowed. Long lines may be broken into pieces by the backslash at last position of the each (except the last) pieces. For example:

```
"a\
b" > "c\
d"
```

means the same as

```
"ab" > "cd"
```

Both the search and the replacement are made up of any combination of the following elements:

1. **Strings** This is a sequence of one or more printable characters. Such sequences of characters are enclosed within matching sets of single or double quotes. If the replacement is on more than one line, each line must be enclosed in its own set of quotes. Any string containing a single quote mark must be enclosed in double quote marks, and any string containing a double quote mark must be enclosed in single quote marks.

2. **Commands or keywords**
   These are short or abbreviated words which instruct the Changes program to perform certain functions. Commands are not enclosed in quotes. Commands must be surrounded by spaces or tabs. The commands are listed in Section 3.4 [Command Description], page 5.

3. **ASCII codes**
   It is sometimes necessary to use non-printing ASCII codes in a MSE table. Both printing and non-printing characters can be represented with ASCII codes; however, it is usually best to simply enclose a printing character in quotation marks rather than use its corresponding ASCII code. For example, it is easier to type and understand *'A'* than its decimal value *d65*.

   A complete chart of ASCII codes has been included in Section 6.4 [ASCII Codes], page 42. Discussion follows on decimal, hexadecimal, and octal codes, respectively.

   The *decimal* ASCII value of the character may be used, *without* quotes, if it is immediately preceded by a *d* (either upper or lower case). For example, *d8* would represent a <BACKSPACE>, and *d9* a <TAB>). The *d* and the ASCII code *must be surrounded by spaces or tabs.* ASCII control codes are listed at the very end of this manual. The decimal codes 0–255 are legal before and after the wedge.

The hexadecimal ASCII value of the character may be used, without quotes, if it is immediately preceded by an "x" (either upper or lower case). For example, x8 would represent a <BACKSPACE>). If and only if hexadecimal is used, only one "x" need precede multiple ASCII codes (eg. X7E08 is tilde and <BACKSPACE>). If you do this, however, be sure that each ASCII code is expressed using 2 digits (eg. tilde and <BACKSPACE> should be represented by x7E08, not x7E8). The hexadecimal codes 0–FF are legal before and after the wedge.

Note: *Octal* may be used by *not* preceding the ASCII code with anything (eg. 10 is <BACKSPACE>). The octal codes 0–377 are legal before and after the wedge.

4. Spaces or tabs

Spaces and tabs separate the strings, commands, and ASCII codes from one another and from the wedge. These spaces and tabs are ignored by MSE during processing, but at least one space or tab is required as a separator.

3.2 How Changes are Processed

Once a match string has been found in the input data, the program does whatever is on the replacement side of the wedge. The matched string is not sent to output unless the replacement side contains a *dup* command or explicitly puts it into the output. Then, instead of continuing to move through the table with the same data, it moves on to the next piece of input data. Data is only processed once, unless it is brought back into the input from the output with the *back(v)* or *backi(v)* command.

3.3 Order of Changes

Change entries are sorted by MSE prior to any processing of the input text. They are sorted according to the number of characters that are being searched for on the left side of the wedge, longest search string first. If the word "sentimental" was in the input file, it would be changed to "emotional," not "sentipeopletal" in the output file, because MSE uses the longest match string (in this case, line 2) first.

```
"men"          > "people"      c line 1
"sentimental" > "emotional"   c line 2
```

If more than one group is being used, the changes are not mixed. Groups will be searched in the order requested, regardless of the length of change strings in any of the other groups. *Within* the groups, however, changes will be searched longest first.

Note that in the following example, line 2 has precedence over line 1:

```
begin > store(affix) "abc" endstore
"test"            > "x"     c line 1
"test" fol(affix) > "y"     c line 2
```

Also, *any(name)* takes precedence over *fol(name)* or *prec(name)*. In the following example, line 2 has precedence over line 1:

```
begin > store(affix) "abc" endstore

"test" fol(affix) > "fol"   c line 1
"test" any(affix) > "any"   c line 2
```

The reason for this is that *fol(name)* and *prec(name)* are *conditions for* the match, not *part of* the match, as opposed to *any(name)*, which is actually *part of* the match (See Chapter 3 [Changes Description], page 3 for a more detailed information on *fol(name)*, *prec(name)*, and *any(name)*).

In general, the following rule is used when MSE is sorting entries: *any(name)* has the same weight as one full character, *fol(name)* and *prec(name)* each have weight of 1/1000th of a

character. This guarantees that entries with *fol(name)* or *prec(name)* will take precedence over similar entries without the *fol(name)* or *prec(name)* (as in EXAMPLE 2), but entries with *any(name)* will take precedence over similar entries with *fol(name)* (as in EXAMPLE 3).

The only time search entries are not sorted by length is when they have the same relative length — for example:

```
begin > store(1) 'aeiou' endstore
'xa' > 'ksa'
'x' any(1) > dup
```

In this case MSE will process the lines in order. When the string "xa" is encountered "ksa" will be output. Even though the next entry would also match the input and appears longer, MSE equates both lines as having the same length. Thus, the "xa" entry remains first in this group and is processed first.

Just to cover all the other cases where ordering of table entries might cause problems, MSE has an 'unsorted' option so that you can completely suppress MSE's sorting. To use this option, place the keyword '`unsorted`' on the begin line as in the following example:

```
begin > unsorted

"a"  > "x"
"ab" > "y"
```

When MSE is processing this table, it will always search the entries in the order they physically appear in the table, thus for the input text "abc" the output would be "xbc" (rather than the output of "yc" which would be expected if the '`unsorted`' option was omitted).

WARNING WARNING WARNING: I STRONGLY discourage use of the '`unsorted`' option because it violates the "longest match first" principle. This can make a table very confusing for a human reader to understand or predict what will happen to his data because it changes the very nature of how MSE operates.

3.4 Command Description

In the list of commands that follows, you will notice that some commands take a parenthesized argument (an example of an argument is the *name* in *store(name)*).

Some commands that take an argument take a value, represented by (v). There are six such commands:

```
back(v)
fwd(v)
omit(v)
backi(v)
prevsym(v)
symdup(v)
```

Other commands that take an argument take a string, represented by name. Any combination of printable characters (including numbers) can be used in a string as an argument for these commands (except a comma, which is used as a separator for multiple arguments).

There are four classes of elements which can be named:

- defined sets of commands
- groups
- storage areas
- switches

The naming of each is totally independent of the naming of any of the others. (eg. nothing automatically happens to switch *examp* when something is stored in area examp).

Any command which takes a parenthesized string argument *name* may take more than one string, separated by commas (i,j,k). This has the effect of repeating the command. For example, *if(1,2)* is the same as *if(1) if(2)*. For some commands (eg., *define*) the use of multiple names is meaningless; for others it could be misleading (eg., *use(1,2)* does not equal *use(1) use(2)*).

add(name) 'number' — ADD number TO STORE name

This command adds the value of number to the value in storage area name. It can only be used on the right side of the wedge. The results of the operation are stored in name, replacing name's previous contents. For example, the following will output "56" :

```
begin  > store(test) '22' endstore
         add(test) '34'
         out(test)
```

Note: A sign (+ or −) may precede the number. Leading zeros in a storage area will be removed after an add (or any other arithmetic operation except *incr* and *decr*). If *store(test)* had"0022" in the above example before the add operation, the final result would still be "56".

any(name) — ANY ELEMENT OF STORAGE AREA name

This function causes a match if any single character in the specified storage area is found in the input data. It may be combined with a string or used alone. This command can only be used on the left side of the wedge. It is useful for matching words which use any element of a closed class (eg., any vowel). In contrast to the *preci*, *prec*, *fol*, and *wd* commands, the character is actually matched and can be output with *dup* or stored in a storage area.

For example, you could change all the consonants to *C* and all the vowels to *V* in a text,then run it through the word count program (`wc`) to get a count of all the word-level CV patterns:

```
begin > store(vowel) 'aeiou' endstore
     store(cons) 'bcdfghjklmnpqrstvwxyz' endstore
     store(punct) '.,"?:;!()[]{}' endstore
any(vowel) > 'V'    c Vowels become V
any(cons)  > 'C'    c Consonants become C
any(punct) > ''     c Remove punctuation
```

This will delete all punctuation, change all vowels to 'V', and all consonants to 'C'.

See also the example under the *fol* command.

append(name) — APPEND TO STORAGE AREA name

This command is quite similar to the *store(name)* command. However, the *store(name)* command causes the previously stored contents of area *name* to be discarded, whereas the *append(name)* command *retains the previous contents* and inserts the new data at the "end," following any data that was already in the storage area. This command can only be used on the right side of the wedge.

back(v) — MOVE BACK v CHARACTERS FROM OUTPUT

This command causes the last *v* characters output to be removed from output or storage and put back into the input stream of text, so it can be checked for a match again.

This command can only be used on the right side of the wedge. The maximum number of characters that can be backed is to the beginning of the storage area (or output). For example:

```
'  ' > ' ' back(1)
```

changes all sequences of spaces to one space, because the space character that has been output can again be a part of the following match.

Be careful when using the *back* command, since it is easy for the table to become hung up in an endless loop. If you use the *back* command, make sure there is either something else in the group that will match the results or send the table to another group with the *use* command. For example:

```
group(1) c make orthographic changes in word entries only
   '\w' > dup use(2)

group(2) c change ae to e and return to group one
   'ae' > 'e'
   '\' > dup back(1)
```

Without a *use* command to get the program out of *group(2)*, the program will hang up when it comes to the next back slash. It will recognize the back slash, dup it, back up, recognize the back slash, dup it, back up ... on and on.

backi(v) — MOVE BACK v CHARACTERS FROM INPUT

This command causes the last *v* inputed bytes to be removed from the history of the input stream and put back into the input stream of data, so it can be checked for a match again.

This command can only be used on the right side of the wedge. The maximum number of characters that can be backed is to the beginning of the input stream history. For example:

```
'  ' > backi(1)
```

does the same as in appropriate example with the *back*.

If you use the *backi* command, consider the same matters as with *back*. The *backi* is like but more difficult than *back*.

begin — BEGINNING OF INPUT FILE OR NESTED BLOCK

If used on the left side of the wedge, this command must be by itself, without quotes around it. The replacement which follows the *begin* command will be executed before any input data is read. It will not be executed again.

On the right side of the wedge, this command is used in conjunction with the *end* command to separate a command or string from other commands or strings. They are primarily used for nesting *if*'s and *else*'s. Note that *if*'s and *else*'s cannot be truly nested without using *begin* and *end*. See the *end* command for an example of this.

The *begin* and *end* commands must be used to tell the program when a string ends for mathematical or comparison operation and the next string begins for data that is to be output. For example:

```
nl > add(count) '2' 'ok' nl  c causes an error
```

This produces an error, because MSE concatenates all the strings. Another example is with comparing strings.

```
'.' > ifeq(fruit) 'apple' 'We have apples.' nl
        else 'We do not have apples.' nl
       endif    c this also will not work
```

Even though the contents of storage area fruit is equal to "apple" the command is comparing it to "apple We have apples." And the program will output the message "We do not have apples."

To overcome this problem, use the *begin* and *end* commands.

```
nl > begin
         add(count) '2'  c this works
     end
     nl
'.' > ifeq(fruit) 'apple' begin
          'We have apples.' nl
      end else
          'We do not have apples.' nl
      endif
```

The table will generate no errors and will produce the correct output.

%

c — COMMENT

This command may occur on a line of its own or on the right side of the wedge (after *c* must be *some spaces*). It is used to indicate comments which explain to the user the purpose of entries in the table. The rest of a line containing a *c* or *%* is ignored by the program.

Although the comment lines are ignored by the program, they are the most important lines in a change table for the user. Comments should be added to the beginning of any table to explain the purpose of the table, the form of the expected input text and include the author's name. This information could be of great value later when trying to figure out how the table works, so modifications can be made. Comments should be added throughout the table to describe the purpose of each group, store, switch and define when there are more than one of each. On tables longer than one page these comments should be grouped together either at the beginning or the end of the table so the user can find them easily.

clear(name) — CLEAR SWITCH name

This command clears (un-sets) a switch which was set by a *set* command.

cont(name) — CONTENTS OF STORAGE AREA name

On the search side, this function causes the contents of the specified storage area to be treated like a match string. For example:

```
begin       > store(quark) "abcd" endstore
cont(quark) > "wxyz"
```

will function exactly like "abcd" `>` "wxyz"

On the replacement side, this function is used in conjunction with the *ifeq(name) 'string'*, *ifneq(name) 'string'*, and *ifgt(name) 'string'* commands. For example:

```
'x' > ifeq(proton) cont(quark) out(quark)
      endif
```

says "if the contents of storage area *proton* equals the contents of storage area *quark*."

decr(name) — DECREMENT STORE name ONE COUNT

This command causes the last character of store *name* to be decremented by one so that it becomes the previous character on the ASCII chart. In the following example:

```
begin > store(zork) 'x' decr(zork) out(zork)
```

the *x* in *store(zork)* is decremented to be a *w*. This command can only be used on the right side of the wedge.

If the last character in the store is a "0," then the next-to-last character in the store will be decremented by one and the "0" will be changed to a nine. In the following example:

```
begin > store(alpha) 'B2'
        decr(alpha) decr(alpha) decr(alpha)
        out(alpha)
```

the output is "A9." Had there only been a "2" in store(alpha), rather than "B2," then MSE final result would have been "9."

The *decr* command is very like to *incr* command. It also preserves leading zeros in a store. For example if store *alpha* contained "0001," it would contain "0000" after doing *decr(alpha)*.

It should be noted that *decr(x)* is not identical to *sub(x) '1'*. The *decr(x)* command will preserve leading zeros, the *sub(x)* command will not. Also, *decr(x)* is allowed on stores which contain non-numeric strings, whereas *sub(x)* is not. The *decr(x)* ignores the sign of the the number.

define(name) — DEFINE SET OF COMMANDS name

This command allows the user to define a set of commands to be executed by the *do(name)* command. Whatever number or name you use for *name* when you *define* the set for the first time is what you must use when you subsequently do it (see the *do(name)* command). The form of the command is:

```
define(name) > commands to be executed
```

As a matter of practice, it is probably best to put all defined commands at the beginning of the table, after the *begin* statement, and before the first group. This command occurs only on the *search side* of the table.

div(name) 'number' — DIVIDE STORE name BY number

This command divides the value in the storage area *name* by the value specified by *number*. The results of the operation are stored in *name*, replacing *name*'s previous contents. For example, the following will output "7" :

```
begin > store(results) '21' endstore
        div(results) '3'
        out(results)
```

Any remainder will be discarded. In the following example, 21 divided by 5 is equal to 4 with a remainder of 1. MSE will discard the remainder "1" and store a "4" in *store(results)*:

```
begin > store(results) '21' endstore
        div(results) '5'
        out(results)
```

do(name) — DO SET OF COMMANDS name

This command causes a set of commands which were specified by a *define(name)* command to be executed. It can only be used on the right side of the wedge. For example:

```
define(vowel) > '***' dup '***'
'a' > do(vowel) set(proton)
'e' > do(vowel) set(neutron)
'i' > do(vowel) set(nucleus)
'o' > do(vowel) set(quark)
'u' > do(vowel) set(fusion)
```

The *do* command is more flexible than the *next* command because do can be used before other commands, and *next* cannot. Also, *do* commands can be "nested," that is, they can be used inside of *define*s. For example:

```
define(1) > 'x' do(2) 'x'
```

```
define(2) > 'y' do(3) 'y'
define(3) > 'z'
'a'       > 'w' do(1) 'w'
```

In this example, an *a*, will be changed to *wxyzyxw*.

In this example, when the command *do(1)* is encountered in the table, it causes MSE to execute the commands following the *define(1)* command. Those commands happen to include a *do(2)* command which cause MSE to execute the commands following the *define(2)* command. Those instructions happen to include a *do(3)* command, which causes MSE to execute the commands following the *define(3)* command. At this point the nesting depth is three. After the commands following the *define(3)* command are finished, MSE will go back and finish the commands (if any) in *define(2)*. When finished doing *define(2)*, MSE will go back and finish the commands (if any) in *define(1)*. When finished doing *define(1)*, MSE will go back and finish the commands (if any) on the line that originally contained the *do(1)* command.

dup — DUPLICATE SEARCH ELEMENT

This function will duplicate the search element of the change into the output file or storage area. Duplication may be done repeatedly.

else — ELSE

The *else* command signals the program to take action when the condition examined by the *if* statement is not true. It also signals the program to stop taking action when the condition examined by the *if* statement is true.

The *else* command is the second part of the three parts of the *if* statement. The first part is the *if* command and the last is the closing *endif* command. The *else* command is optional.

```
'I will ' > dup
            if(rain) 'stay inside.'
            else 'go for a walk.'
            endif
```

Putting this MSE table into English would give, "I will stay inside if it is raining, otherwise I will go for a walk."

Note how the *if*, *else*, and *endif* commands were aligned. This is not necessary for the table to function, but makes it easier to see what action will take place when the condition is true or false. And shows that there is an *endif* command to terminate the *if* condition.

If you must check on multiple conditions, you should use the *begin* and *end* commands for nesting. See the *end* command for an example of this. See also *if(name)*, *ifeq(name)*, *ifgt(name)*, *ifn(name)*, *ifneq(name)*.

end — END OF NESTED BLOCK

This command indicates the end of a block of nested *if*s or *else*s. The corresponding block initiator is the *begin* command. (Do not confuse the *begin* command which initiates a nested block with the *begin* command which allows commands to be executed at the beginning of a file.) For example:

```
'x' > if(1) begin
          if(2) 'a'
          else 'b'
      end else begin
          if(2) 'c'
          else  'd'
```

```
      end
c The preceding entry outputs a if 1 and 2 are on,
c   b if 1 is on and 2 off, c if 1 is off and 2 on,
c   and d if 1 and 2 are both off
```

End can also indicate the end of a repeated block of commands. (See the *repeat* command.)

endfile — END OF INPUT FILE

The replacement for this command will be executed after all input data has been read and processed. This command must be listed as the only element of a search. For example:

```
endfile > out(3)  c store 3 out at very end
```

This *endfile* entry may occur at any point in the table, it does not have to be last.

endif — END IF

This command marks the end of a conditional segment of a replacement specification. It applies to all conditionals currently in effect, unless nested with the *begin* and *end* commands. See also *if(name)*, *ifeq(name)*, *ifgt(name)*, *ifn(name)*, *ifneq(name)*.

endstore — END STORING

This command will cause any storing in effect to stop. It reroutes the output from storage to the actual output file. See *store(name)*.

excl(name) — EXCLUDE GROUP name

This command will exclude the group *name* (see the *group(name)* command) from the groups that MSE is currently using. It can only be used on the right side of the wedge. This table:

```
begin > use(dos,mac,unix,windows)
        use(dos,unix,windows)
```

will have the same effect as this table:

```
begin > use(dos,mac,unix,windows)
        excl(mac)
```

It has the opposite effect of the *incl(name)* command.

fol(name) — MATCH IF FOLLOWED BY ANY CHARACTER IN name

This function will cause the string to be matched only when *followed* by any one of the characters contained in the storage area *name*. Note that the character itself is not matched and will not be output by the *dup* command. The character is a *condition of* the match, not a *part of* the match.

This function should be used only on the search side. It is particularly convenient for matching strings which are required to be at the end of a word. All word-final punctuation, including space, can be stored in a particular storage area and used with the *fol(name)* command. Here is an example of the *fol* command:

```
begin > store(vowel) 'aeiou' endstore
        store(stop) 'bdg' endstore
any(vowel) fol(stop) > dup dup
```

This would double any character found in storage area *vowel* (a, e, i, o, or u) that was followed by one of the characters in storage area *stop* (b, d, or g).

More than one *fol(name)* command may be used in succession. For example, the command *'test' fol(1,2,3)* will look for *test*, followed by something in storage area *1*, followed by something in storage area *2*, followed by something in storage area *3*.

Compare this with the commands *wd(name)*, *prec(name)*, and *any(name)*.

fwd(v) — MOVE FORWARD v CHARACTERS

This command causes the next *v* characters that would be input to be passed directly to output or storage, without being considered for matching in the table. This command may only be used on the right side of the wedge.

group(name) — GROUP OF CHANGES name

This command identifies the following changes as belonging to the group *name*. Which group of commands is currently active is controlled by the *use(name)*, *incl(name)* and *excl(name)* commands. If a table consists of only one group, the *group* command is *not* necessary. Whatever name you choose for *name* in the *group(name)* command is the name you must specify when you subsequently make the group active with the *use(name)* command.

This command is put at the beginning of a line by itself and is *not followed by a wedge*. In a sense, it is not a command, but a label at the beginning of each set of change entries.

Groups are particularly useful when certain changes are wanted in one context but not in another, eg., changing the orthography of one language inside a bilingual dictionary file.

If the change table is longer than two pages, then numbers should be used for each group name instead of string names. This will make it easier for the user to follow the flow of the table when the program changes groups. The program will always start with the group that has a name of "1" or the first group in the table, if there is no *group(1)*. However, a *use* command in the *begin* statement can initialize the table to start with a specific group or groups active.

If a number followed by letters is used as the name of a group, that number is associated with that group name when the table is loaded. Should that name begin with the numeral 1, that group will function as though it were group(1) and will become the first active group in the table unless otherwise designated. For example:

```
group(main)
'\w' > dup use(1st)
'\d' > dup use(1st)

group(1st)
'a' > 'V'
'b' > 'C'
'\' > dup back(1) use(main)
```

MSE will begin processing text using the set of changes found in *group(1st)* not in *group(main)* as you might expect. This feature of MSE could cause strange looking output, until the table gets in synch with the data coming into it.

if(name) — IF SWITCH name IS SET

The *if* command checks the status of switch *(name)* and executes the following commands based on the condition of the switch. The *if* command can only be used on the right side of the wedge.

If the switch is set, then following replacement commands are executed. (See *set(name)* and *clear(name)* commands). If the switch is clear, then the following replacement commands are ignored.

There are three parts to an *if* command, and the *if(name)* is the first part. The last part is the *endif* command. The second part (optional) is the *else* command.

The *if* command may be nested with other *if* commands (checking for multiple conditions) by using the *begin* and *end* commands. (See *end* command for an example of nested *if* commands.) See also *ifn(name)*.

ifeq(name) 'string' — IF STORE name EQUALS string

This command executes the following commands if the content of store *name* exactly matches the string. It can only be used on the right side of the wedge. The sequence *'string'* is any combination of literal strings and ASCII characters (such as d8 for backspace). The *cont(name)* command can be used instead of a string to compare the contents of store *name* to the contents of another storage area. (See the *cont(name)* command.) For example:

```
'x' > ifeq(orange) 'apple' set(ripe)
      endif
```

will have exactly the same results as:

```
'x' > store(fruit) 'apple' endstore
      ifeq(orange) cont(fruit) set(ripe)
      endif
```

The string is terminated by the following command.

The conditional execution is terminated by an *endif* or *else* command, or by the end of the table entry. MSE does a byte-for-byte ASCII comparison.

See also the discussion under the *if(name)* command.

ifgt(name) 'string' — IF name IS GREATER THAN string

This command is very similar to the *ifeq* command, in that it compares the contents of store name to the following string or to the contents of another storage area. It can only be used on the right side of the wedge. If the contents of *name* are "greater than" the *string*, the following commands will be executed; if not, they will be skipped.

When comparing characters, "greater than" is strictly according to ASCII codes. See the ASCII table at the end of this manual for details. Thus, *abc* is greater than *b* (*abc* has length greater than *b*) and *a* is greater than *B* or *1*.

ifn(name) — IF SWITCH name IS NOT SET

This command is completely parallel to *if* except that it executes the commands and replacements following it if the switch is *not* set (is clear), and doesn't execute it if the switch *is* set.

ifneq(name) 'string' — IF name IS NOT EQUAL TO string

This command is like *ifeq*, except that the following commands and replacements are executed if *name* is *not* equal to the string.

incl(name) — INCLUDE GROUP name

This command will include group *name* (see the *group(name)* command) with the group(s) that MSE is currently using. It can only be used on the right side of the wedge. This table:

```
begin > use(2,5,8,4)
```

will have the same effect as this table:

```
begin > use(2,5,8) incl(4)
```

This command has the opposite effect of *excl(name)*.

incr(name) — INCREMENT STORE name ONE COUNT

This command causes the last character of store *name* to be incremented by one so that it becomes the next character on the ASCII chart. In the following example:

```
begin > store(zork) 'x' incr(zork) out(zork)
```

the *x* in *store(zork)* is incremented to be a *y*. This command can only be used on the right side of the wedge.

If the last character in the store is a "9," then the next-to-last character in the store will be incremented by one and the "9" will be changed to a zero. In the following example:

```
begin > store(alpha) 'A7'
        incr(alpha) incr(alpha) incr(alpha)
        out(alpha)
```

the output is "B0." Had there only been a "7" in store(alpha), rather than "A7," then MSE final result would have been "0."

A common use of *incr(name)* is to count the number of occurrences of a certain character or string in a file:

```
'x' > dup incr(total)      c count every x
endfile > out(total)       c output count
```

The above table will count every occurrence of *x*. The *incr* command preserves leading zeros in a store. For example if store *x* contained "0001," it would contain "0002" after doing *incr(x)*.

It should be noted that *incr(x)* is not absolutely identical to *add(x) '1'*. The *incr(x)* command will preserve leading zeros, the *add(x)* command will not. Also, *incr(x)* is allowed on stores which contain non-numeric strings, whereas *add(x)* is not. The *incr(x)* ignores the sign of the the number.

mod(name) 'number' — REMAINDER OF STORE name DIVIDED BY number

This command divides the value in the specified storage area by the value of number. The remainder from the division operation is stored in *name*, replacing name's previous contents. For example, the following will output "7" :

```
begin > store(test) '40' endstore
        mod(test) '11'
        out(test)
```

Since 40 divided by 11 is 3 with a remainder of 7, MSE discards the 3 and stores the 7 in storage area *test*. If there is no remainder, MSE will store a 0 as the remainder. This command can only be used on the right side of the wedge.

mul(name) 'number' — MULTIPLY STORE name BY number

This command multiplies the value in the specified storage area by the value in *number*. The results of the operation are stored in *name*, replacing *name*'s previous contents. For example, the following will output "48":

```
begin > store(1) '4' endstore
        mul(1) '12'
        out(1)
```

This command can only be used on the right side of the wedge.

(name) — NAME OF STORAGE, SWITCH, GROUP, OR DEFINE

Any combination of printable characters (including numbers) can be used in the name to designate specific switches, groups, defines or storage areas. The only exceptions are a space, a right round bracket, and a comma. A comma is used as a separator for multiple designators. The naming of each is totally independent of the naming of any of the others. (eg. nothing happens to switch *examp* when something is stored in area *examp*).

Any command which takes a parenthesized string argument name may take more than one string, separated by commas (i,j,k). This has the effect of repeating the command. For example, to clear three storage areas, *store(1) store(2) store(3) endstore* is the same as *store(1,2,3) endstore*. The only exception is the *use* command,

in which *use(1,2)* makes both *group(1)* and *group(2)* active, while *use(1) use(2)* makes only *group(2)* active.

next — USE REPLACEMENT IN NEXT ENTRY

This command executes the replacement side of the next search entry. This is useful when a number of similar match-strings need the same change. It saves table space and makes the table easier to read. For example:

```
'a' > next   c change all vowels to V
'e' > next   c  and add one to vowel count
'i' > next
'o' > next
'u' > 'V' incr(vowel)
```

Commands and replacement strings may precede *next* on the replacement side, but anything following the *next* command on that replacement is ignored.

See also the *define* and *do* commands.

nl — NEW LINE

If used on the left side of the wedge, *nl* matches an `Enter` keyed in the input. If used on the right side of the wedge, it has the effect of putting an <ENTER> into the output. Note: It's difficult to put a ASCII code of <ENTER> between quotes. The only simple way to indicate an <ENTER> is to use nl.

not(name) — LOGICAL NEGATE SWITCH name

This command logical negates a switch or flag which you can check for conditional execution of table entries. It can only be used on the right side of the wedge. See also *clear* and *set* commands.

'' — NULL MATCH or REPLACEMENT

If used on the left side of the wedge, the null match will match when nothing else will. Note that the following restriction must be observed to avoid putting the table into a loop:

When '{}' is used on the left side of the wedge, you should put either a *fwd(v)* or an *omit(v)* command or a *use(name)* command on the right side of the wedge so progress can be made through the input (see the *fwd(v)*, *omit(v)*, and *use(name)* commands). Since '{}' matches when the next character in the input file doesn't match anything, that character must be removed to allow the possibility of matching the next character. The commands *fwd(v)* and *omit(v)* accomplish this. The *use(name)* command sends the program to a different set of matches, where the character might match. For example:

```
'a'    > 'a'
''     > fwd(1) '-'  c this puts a hyphen after
                     c any char other than 'a'
```

The '{}' is meaningless when used on the right side of the wedge. It is sometimes used, however, to visually signify that nothing is being output. It is not necessary, but is helpful to clarify what is happening. (Its absence does not save any table space.) Thus, the following:

```
"a"    > ''    c get rid of every a
"b"    > "c"   c change every b to c
```

is the same as:

```
"a"    >       c get rid of every a
"b"    > "c"   c change b to c
```

omit(v) — OMIT v CHARACTERS FROM INPUT

This command causes the next *v* characters that would be input, to be discarded. These characters will not be passed through the table to be matched nor put into output or storage. This command can only be used on the right side of the wedge.

out(name) — OUTPUT STORAGE AREA name

This command stops any storage in progress and sends the contents of storage area *name* to the output. The contents of *store(name)* remain unchanged and may be output more than once. Unless there is another *store* command, all results will then go to the actual output. This command can only be used on the right side of the wedge.

The *out* command closes any storage area that may be open, and output continues to be routed to the actual output after the command is executed.

outs(name) — OUTPUT STORE name EVEN WHILE STORING

This command is the same as the *out* command, except that it continues any storing already in progress. It can only be used on the right side of the wedge. This allows transfer of material between storage areas. For example, the following copies the contents of storage area *1* to storage area *2*, and storage area *2* remains open after the *outs(1)* command is executed:

```
store(2) outs(1)
```

Note that the content of storage area *1* does not change.

prec(name) — MATCH IF PRECEDED BY ANY CHARACTER IN STORE name

This function will cause the string to be matched only when that string is *preceded* by anyone of the characters contained in the specified storage area. This command can only be used on the search side. Note that the character itself is not matched and will not be output by the *dup* command. The character is a *condition of* the match, not a *part of* the match.

This function is particularly convenient for matching strings which are required to be at the beginning of a word; any character that may appear before a word such as a space, can be stored in a particular storage area and used with *prec(name)* command. An example of the *prec* command follows:

```
begin > store(begin-word) ' ' nl '<"([{' endstore
'c' prec(begin-word) > 'ch'
```

This would change any *c* that is preceded by a word-break character to the character sequence *ch*.

More than one *prec(name)* command can be used in succession. For example, *'test' prec(1,2,3)* will look for *test*, preceded by something in storage area *3*, preceded by something in storage area *2*, preceded by something in storage area *1*.

Compare this with the commands *wd(name)*, *fol(name)*, *preci(name)*, and *any(name)*.

preci(name) — MATCH IF PRECEDED BY ANY CHARACTER IN INPUT name

This function will cause the string to be matched only when that string is *preceded* in the input stream history by anyone of the characters contained in the specified storage area. It is very like to *prec*, but in several cases is more useful. For example:

```
begin > store(delims) ' ' nl '<"([{}])' endstore
'i' preci(delims) > '[I]'
```

This would change any *i* that is preceded by a word-break character to the character sequence *[I]*. In the case of using *prec* in this example not only the first letter *i* in a word will changed but all the consequent letters *i* too.

prevsym(v) — MATCH IF v-th PREVIOUS SYMBOL IS THE SAME

This command can only be used on the left side of the wedge. For example:

```
begin > store(a) '123' endstore
any(a) '5' prevsym(2) > '*'
    c changes '151', '252', '353' to '*'
```

Number *v* is the offset from the position of the current *prevsym* (it's 0) to the left in the matched string.

read — READ FROM KEYBOARD

This command reads a line from the keyboard into the current store if storing, or directly into output. It can only be used on the right side of the wedge. MSE stops reading characters from the keyboard when the `Enter` key is pressed. The `Enter` is simply a signal to the *read* command to stop reading characters from the keyboard; the <ENTER> does not actually go to the storage area or output.

Prior to issuing a *read* command, it would be advisable to use the *write* command to write a message on the screen so that the person at the keyboard would realize that the computer has paused and is waiting for input from the keyboard.

repeat — REPEAT FROM begin

This command goes back to the nearest *begin*. This command can only be used on the right side of the wedge. For example:

```
   c This table fills short lines with the letter x
   c  until all lines have sixty characters
begin > store(char) ' abcdefghijklmnopqrstuvwxyz,.?!'
        store(count) '0' endstore

any(char) > dup add(count)'1'
nl > ifgt(count) '59' begin
         '**** ERROR count 60 or greater ***' nl
     end else begin
         add(count)'1'     c Increment count
         'x'               c and output an x
          ifneq(count) '60' c If count not sixty,
              repeat        c   go back to begin
          endif
          store(count) '0' endstore
          nl     c restore count and output newline
     end endif
```

Be careful when using the *repeat* command. In this example we checked first to make sure that the count was less than 60 before starting the *repeat* command. If, for some reason the count was 60 or greater when we encountered a newline, the program would hang up in an endless loop. Always check whatever is being used to control the *repeat* command to make sure that it is set properly before beginning the *repeat* loop.

It may be easier to run MSE twice (pass the data through two different change tables) than make a complex table to do everything in just one pass.

set(name) — SET SWITCH name

This command sets a switch or flag which you can check (using *if* commands) for conditional execution of table entries. It can only be used on the right side of the wedge. Whatever number or name you use when you *set* it for the first time is what you must use when you subsequently *clear* or test the flag.

A switch can be "turned off" by using the *clear* command. All switches are initially clear (not set).

store(name) — STORE IN STORAGE AREA name

This command reroutes the output to an internal storage area. It can only be used on the right side of the wedge. Whatever you call the storage area in the *store(name)* command is what you must use when you subsequently output its contents (see the *append*, *out*, and *outs* command as well as description of *name*).

Any data previously stored in the specified area is discarded when a new request to store is given, and any storage being done in another area is stopped.

Storage areas can only be cleared by a *store* command followed immediately by an *endstore*, *out*, or another *store* command. Note that if multiple stores are requested at once, the effect will be to erase and close each until the last, which will be cleared, but remain open to be stored into. The following three lines are equivalent to each other:

```
'x' > store(1,2,3)
'x' > store(1) store(2) store(3)
'x' > store(1) endstore store(2) endstore store(3)
```

sub(name) 'number' — SUBTRACT number FROM STORE name

This command subtracts the value specified by *number* from the value in the storage area *name*. It can only be used on the right side of the wedge. The *difference* is stored in *name*, replacing *name*'s previous contents. For example, the following will output "3":

```
begin > store(value) '17' endstore
        sub(value) '14'
        out(value)
```

symdup(v) — DUPLICATE v-th SYMBOL IN SEARCH ELEMENT

This command can only be used on the right side of the wedge. For example:

```
'abcd' > symdup(0) symdup(2)
    c changes 'abcd' to 'ac'
```

Number *v* is the position in the current search element (counted from 0).

use(name) — USE CHANGES IN GROUP name

This command specifies which groups of changes are currently available to be matched (see the *group(name)* command). Any previous *use(name)* command is cancelled. This command can only be used on the right side of the wedge. If *use(x)* is specified, then the changes in *group(y)* are ignored. For example:

```
group(def)
'\w' > dup use(word)   c change a to aa following
'a'  > 'aa'            c  a \d but not following
group(word)            c a \w marker
'\d' > dup use(def)
```

Several groups can be made available for searching at the same time. For example, *use(1,6,8,4)* causes groups *1*, *6*, *8* and *4* to be searched. The *use(name)* commands do not take effect until *the end of the entry* in which they were specified.

wd(name) — MATCH ONLY IF WORD

This command causes a string to be considered matched only if it is both preceded and followed by any character contained in storage area *name*. Note that the preceding and following characters are *not* considered part of the match and would not be output by a *dup* command. For example, the following table:

```
begin  > store(punct) nl ' .,"()' endstore
'and' wd(punct) > 'also'
```

will change any of the following:

```
and   and.  and,  and''  and(  and)  and<ENTER>
 and  .and  ,and  ``and  (and  )and  <ENTER>and
```

This command is used only on the search side of the table. `wd(n)` means the same as the sequence of command `prec(n) fol(n)`.

Note: When storing the word boundary punctuation, do not include any diacritics. Also keep in mind that there is a small gain in speed if the most frequently used characters are listed first.

write 'string' — WRITE string TO SCREEN

This command writes on the screen the contents of the *string*. A *string* is any combination of literal strings, *nl* commands, and ASCII characters (such as 10 for <BACKSPACE>). This command can only be used on the right side of the wedge. The string is terminated by the following command or next search entry. It may contain *nl*s and multiple lines. For example:

```
'cat' > write nl 'cat found' nl
'bird' > write nl 'feathered friend found' nl dup
```

When *cat* is matched the program writes the message *cat found* on the screen. The screen message is terminated by the next search entry. When *bird* is matched the program writes the message *feathered friend found* on the screen. The screen message is terminated by the *dup* command and *bird* is written to the output, but not to the screen.

wrstore(name) — WRITE STORAGE AREA name TO SCREEN

This command writes on the screen the contents of store *name*. It can only be used on the right side of the wedge. Combining the example under *write* with the example under *incr(name)*, if a count of every x was kept in storage area count, the total could be printed to the screen as follows:

```
endfile > write 'There were '
    wrstore(count) write " occurrences of x" nl
```

3.5 I/O Options

When you type MSE at the console prompt, MSE will wait for the input data (the input will finished after *Ctrl+d*). For example:

```
$ mse
abcd
dcba
Ctrl+d
```

Cause lines *abcd* and *dcba* appearance in the output.

3.6 Command Line Options

When you type MSE at the console prompt, MSE uses default values for change table, input, output, and log streams. You can set your values when you type MSE by command line options. For example:

```
------------------------------------------------------------
$ mse -t - Enter
'a' > 'b' Enter
'b' > 'c' Enter
```

```
Ctrl+d abcd Enter
dcba Enter
Ctrl+d bccd
dccb
$ _
------------------------------------------------------------
```

In this example lines *bccd* and *dccb* appeared in the output. If both the change table and data input stream go from standard input device the change table is entered first. Another example:

```
------------------------------------------------------------
$ mse -t changes.mse -o result.txt data.txt Enter
$ _
------------------------------------------------------------
```

The -t indicates that the table name will follow and -o indicates that the output file name will follow. The input files names don't get preceded by anything. Options may be typed in any order. It's possible to use several input data sources (any combination of the files and standard input). There will only be one output file, however.

In fact, you don't even have to remember this. If you type `mse -h` at the console prompt, MSE will display a list of which "-" goes with which file name.

Summary of Command Line Options

`-t`	Change Table name (default is empty table)
`-o`	Output file or device (default is standard output device)
`-l`	Log file name (default is no logging)
`-`	After `-t` option or in the place of the input filename means standard input device, after `-o` or `-l` options means standard output device
`-V`	Invoking MSE with this option prints its version
`--help` `-h`	Invoking MSE with any of these options prints command line options summary

4 Advanced Features

4.1 Storage Commands

There are five commands directly connected with the storage feature of the MSE program:

```
store(name)
append(name)
endstore
out(name)
outs(name)
```

Some secondary commands which use storage areas, but which are not described in this section, are:

```
add(name)    any(name)    cont(name)
decr(name)   div(name)    excl(name)
fol(name)    ifeq(name)   ifgt(name)
ifneq(name)  incl(name)   incr(name)
mul(name)    preci(name)  prec(name)
sub(name)    wd(name)     wrstore(name)
```

More information on these can be found in Section 3.4 [Command Description], page 5. The expression *(name)* represents any logical name you choose. A logical name can consist of alphabetic characters or numbers, and cannot include spaces, commas or a right parenthesis. The names can be any length. These rules also apply to the names for groups, switches, and defines.

What *store(name)* Does

When the *store(name)* command is encountered, the storage area assigned to that name is first cleaned out — any data stored there previous to encountering the *store(name)* is discarded, without warning. Before using the *store(name)* command, be sure you do not need anything that may be in the storage area. Now rather than send data to the normal output, the data is sent to the temporary store area. Data will continue to be stored in this area until the program encounters another command that affects storage (*append*, *endstore*, *out*, or *outs*).

If storage had been requested to one area *(name1)*, but it is now requested to a different area *(name2)*, the output is diverted to the second area and no longer goes into the first. Only one storage area at a time accepts data.

What *append(name)* Does

The *append(name)* command is quite similar to the *store(name)* command, except the *store(name)* command causes the previous contents of area *(name)* to be discarded. The *append(name)* command retains the previous contents and inserts the new data into the storage area following any data that already was in that storage area.

What *endstore* Does

When an *endstore* command is encountered, any storage that was going on is stopped and output is directed to the normal output, as it does when storage is not requested. Data that is currently in storage will remain there until the program encounters a command (*store*, *append*, *out*, or *outs*) naming that area. Note that no name is required for the endstore command.

By the way, if you want to deliberately clear out the contents of a storage area, the combination of commands *store(name) endstore* will clear it out without affecting output at all.

What *out(name)* Does

When *out(name)* is encountered, two things happen. First, if storage is being done, it is stopped as if an endstore had been encountered. Second, the contents of storage area *(name)* are sent to the output file. Note that no matches are performed; the contents of the storage area do not pass through the change table. Also note that storage area *(name)* is not cleared out; it still contains what it contained before the *out(name)* was encountered. Storage area *(name)* may be output any number of times. If there is nothing in the storage area, nothing is output.

What *outs(name)* Does

The *outs(name)* command is very similar to the *out(name)* command except the *outs(name)* command does not stop storing. This provides a way to transfer data from one storage area to another. This applies whether storing is being done with the *store(name)* command or with the *append(name)* command. For example, to copy the contents of *store(first)* to *store(second)*, use the command *store(second) outs(first) endstore*.

To put the contents of storage areas *first*, *second*, and *third* all together into area four:

```
store(four) outs(first) outs(second) outs(third) endstore
```

or

```
store(four) outs(first,second,third) endstore
```

An Example of Storage

The storage feature has a number of uses. Frequently it is used when the user wants the output in a different order than the input order. The following example illustrates the use of storage in a simple dictionary reversal.

Let's suppose that you had a huge text file that was a bilingual dictionary that a Spanish speaker would use to find the meaning of English words. A text file for such a dictionary might be keyed in with each line preceded by a SIL Standard Format marker as follows:

```
\w word in English
\p part of speech in Spanish
\d definition in Spanish
\i illustrative sentence in English
\t translation of illustrative sentence in Spanish
```

Let's suppose now that you wanted a dictionary that would go the other way, to allow an English speaker to find the meaning of Spanish words. We could use the first dictionary as a basis for our new dictionary, creating a MSE table to rearrange things for us.

```
Sample of Input (before):      Desired Output (after):
  \w cat                         \w gato
  \p n                           \p n
  \d gato                        \d cat
  \i The cat is black.           \i El gato es negro.
  \t El gato es negro.           \t The cat is black.
```

Note that the word and definition "changed places," as did the illustrative sentence and its translation. (For the moment, we will not deal with the fact that different abbreviations would probably be used for the part of speech — we are interested in the process of the reversal.) The following table is what is needed for a reversal.

```
-------------------------------------------------------------
  "\w " > out(def,part,word,trans,ill)
                 c output reversed entry
          store(trans,ill,def,part,word)
                 c clear storage areas
                 c   and store entry word
          "\d "  c mark word as definition

  "\p " > store(part) "\p "   c keep as part of speech
  "\d " > store(def)  "\w "   c mark def. as entry word
  "\i " > store(ill)  "\t "   c mark illus. as
                              c    translation
  "\t " > store(trans) "\i "  c translation as
                              c    illustration.

  endfile > out(def,part,word,trans,ill)
                              c output last entry
-------------------------------------------------------------
```

What does this say? It is easier to understand if we look at it in pieces.

Conceptually, the first thing to do is to store everything that comes in, in different storage areas. If you look closely, you will see the following in the above table, among other things.

```
"\w " > store(word)     c store entry word
"\p " > store(part)     c store part of speech
"\d " > store(def)      c store definition
```

```
"\i " > store(ill)      c store illustrative sentence
"\t " > store(trans)    c store translation
```

The data comes in and the \w is found. Storage area *(word)* is requested. Data that follows passes through the table unchanged. However, it does not go to the output file; it is sent into storage area *(word)*. When the \p comes through, it matches and storage area *(part)* is requested. Data that follows is sent into storage area *(part)*, and so forth.

Soon a \w is found again, and that is where some of the other commands in the table really take effect. Let's look more closely at the \w entry, as it really is in the table.

```
"\w " > out(def,part,word,trans,ill)
                        c output reversed entry
        store(trans,ill,def,part,word)
                        c clear storage areas
                        c   and store entry word
        "\d "           c mark word as definition
```

The first line of it:

```
"\w " > out(def,part,word,trans,ill)
                        c output reversed entry
```

says to stop any storing that may be being done, and to output the data in the storage areas in the order: *def*, *part*, *word*, *trans*, *ill*. As you may recall, the definition was stored in area *(def)*. That is output first. The part of speech is in storage area *(part)* and it is output second. The main entry word is in storage area *(word)* and it is output third. And so forth. Comparing this to the desired output, it is indeed what is wanted. The next line:

```
store(trans,ill,def,part,word)  c clear storage areas
                                c and store entry word
```

is a bit more obscure. It is perhaps easier if we look at an equivalent set of commands:

```
store(trans) store(ill) store(def) store(part) store(word)
```

This has exactly the same effect as *store(trans,ill,def,part,word)*. Requesting storage into an area causes its current contents to be discarded. If another storage area is immediately requested, nothing is stored in the first. Thus, the command *store(trans)* says, "stop storing any place else, erase anything that might be in storage area *(trans)* and begin storing something new there." This is immediately followed by *store(ill)* which says, "stop storing any place else, erase anything that might be in storage area *(ill)* and begin storing something new there." What happened? The effect was to erase anything in storage area *(trans)* without putting anything new there. Similarly, since the command *store(def)* follows immediately, storage area *(ill)* has been erased and nothing new put there. This continues until at last the *store(word)* command is encountered. By the way, we could have said:

```
store(trans) endstore
store(ill)   endstore
store(def)   endstore
store(part)  endstore
store(word)  endstore
store(word)
```

This would have had the same effect as *store(trans,ill,def,part,word)*.

Since no other store or endstore command follows the *store(word)* command, something actually can be stored in area *(word)*. And, in fact, that happens immediately. The line:

```
"\d "             c mark word as definition
```

will be stored in area *(word)*.

In general, this is what is happening:

```
"\w " > "\d "
"\p " > "\p "
"\d " > "\w "
"\i " > "\t "
"\t " > "\i "
```

If you compare the Sample of Input with the Desired Output, you will notice that the markers change. What was marked as the main entry word is now marked as the definition, and vice versa. This type of changing is one of the most basic features of the MultiStream Editor program. A sequence of characters is matched and is replaced by another sequence of characters. Putting the marker changes together with the storage, the table has:

```
"\w " > store(word)  "\d "    c mark word as definition
"\p " > store(part)  "\p "    c keep as part of speech
"\d " > store(def)   "\w "    c mark def. as entry word
"\i " > store(ill)   "\t "    c mark illus. as
                              c    translation
"\t " > store(trans) "\i "    c mark translation as
                              c    illustration
```

The entry word "cat" goes into storage area word. The new marker "\d" should go there too — before the word "cat" does — just as the old marker "\w" was before the word "cat." Thus, the new marker should be the first thing stored in the storage area. In order for *\d* to be stored, it must follow the store command, not precede it. Because the *\d* follows the *store(word)*, it is stored immediately in area word. Then the end of the command is encountered. The rest of the data between the "\w" and the "\p cat" is not changed because it matches nothing in the table. It would have gone to the output file, but because storage has been requested, it goes into storage area word — where the *\d* already is. When the "\p" is encountered, the change table calls for storage to be switched to area part. Then the "\p" is "changed" to "\p," and sent to storage area part. The same process is followed for the other parts of the data.

How can we tell when we have stored all there is of a given entry and that we are starting a new word in the dictionary? — when we get to the beginning of the next entry. That is why the out command is at the beginning of the "\w" entry. Another way to know that we have just finished storing a given entry is when we reach the end of the input file. In the following part of the table, the *endfile* on the left of the wedge means "Do this when we get to the end of the input stream:"

```
endfile > out(def,part,word,trans,ill)   c output last
                                         c   entry
```

The command *endfile* means: at the very end of the data, when everything has been looked at, but before the program stops, to output the last reversed entry, just like the "\w" entry in the table does — but there won't be another "\w" coming. That is the only way to tell the program that it is done. The *endfile* on the left of the wedge catches the end of file mark in the data. The *endfile* on the right side tells the program to send an end of file mark to the output file, close the output file, stop processing and return to the console prompt.

There are a few other comments that need to be made about the table. For convenience it is reproduced below:

```
"\w " > out(def,part,word,trans,ill)
                          c output reversed entry
        store(trans,ill,def,part,word)
                          c clear storage areas
                          c   and store entry word
        "\d "             c mark word as definition
"\p " > store(part) "\p "   c keep as part of speech
```

```
"\d " > store(def)  "\w "   c mark def. as entry word
"\i " > store(ill)  "\t "   c mark illus. as
                           c    translation
"\t " > store(trans) "\i "  c mark translation as
                           c    illustration
endfile > out(def,part,word,trans,ill)
                           c output last entry
```

When the first "\w" is encountered, the program executes the *out(def,part,word,trans,ill)* command. This is no problem, because when nothing is stored in a storage area, nothing is output.

There is no problem if, for example, some entries do not have illustrative sentences or apart of speech. Why? At the beginning of each new entry, all the storage areas are completely erased. Nothing is stored in an area unless the marker for that area is found in the data. Thus, the following input would produce the following output:

```
input                          output
\w cat                         \w gato
\p n                           \p n
\d gato                        \d cat
\i The cat is black.           \i El gato es negro.
\t El gato es negro.           \t The cat is black.
\w dog                         \w perro
\p n                           \p n
\d perro                       \d dog
\w mouse                       \w raton
\p n                           \p n
\d raton                       \d mouse
 .                              .
 .                              .
 .                              .
```

If we had not used the *store(trans,ill,def,part,word)* command to erase the storage areas, the illustrative sentences for the "\w cat" entry would also have been printed out with the "dog/perro" entry and following entries, until a new set of illustrative sentences in the input was encountered. Whenever you see such results, you can be sure that some storage area has not been cleared.

To get rid of blank lines before entries, add the following:

```
nl "\w" > next
```

just before the "\w" entry as it is. To output blank lines, modify the "\d" entry to read:

```
"\d" > store(def) nl "\w"
```

This will put a blank line in front of the very first record, but that should not be a problem. Another way of dealing with blank lines is described in Section 4.2 [The Back Command], page 25.

4.2 The Back Command

The command *back(v)* pulls back the previous number of characters which were stored or output and treats them as if they were new input.

One of the main difficulties in writing a general change table is trying to anticipate all their regular typing sequences — both legitimate variations as well as "errors" — which occur in any manuscript.

An often-encountered problem is that of extra spaces or <ENTER>s. These occur in various combinations and in varying numbers. Often in a printout, it is desirable that a sequence of spaces be treated like one. (There are exceptions, of course.) Without the *back* command there is no way to effectively do so.

The following command line causes any sequence of spaces to become one space:

```
"  " > " " back(1)
```

This says, "if there are two spaces, put out one instead; then put that one character back into the input so that it is available to be matched again in the table." Keep in mind that the "1" in *back(1)* does not refer to a storage area, but to the quantity of characters to be moved back. See Section 3.4 [Command Description], page 5 for more information on the *back* command.

If the space is followed by another space — ie, if there were originally three spaces in a row — then the space that was output and backed over, plus the space following, will be a pair of spaces which will match the entry above and be reduced to one space. This will continue for however many spaces occur together. Finally just one space will be left.

Another place stray spaces occur is at the ends of lines. The automatic wrap feature of various edit programs removes such spaces, but people still manage to get a few. The following command will remove a space that precedes an <ENTER>. (Multiple spaces preceding the <ENTER> will have been reduced to one by the command described earlier.)

```
" " nl > nl back(1)
```

This says, "if a space immediately preceding a new line is matched, output a new line; then back up so the new line is treated like input and is available to be matched again in the table." Yet another source of multiple spaces in printouts is blank lines — multiple <ENTER>s. The following command takes care of blank lines in the same way multiple spaces are taken care of.

```
nl nl > nl back(1)
```

And the following command removes spaces from the beginnings of lines, just as a previous one removed them from the ends:

```
nl " " > nl back(1)
```

Together these four force any sequence of spaces to be treated like a single space, and any combination of spaces and <ENTER>s to be treated like a single <ENTER>.

But that is only one aspect of the difficulties needing to be dealt with. It is not uncommon to find text files that include SIL Standard Format markers or TEX markers. These markers most commonly take the form of a "\" followed by a lower case character and then a space character. Markers are put in by the person editing the file so that the file can be manipulated later by utility programs (such as MSE). People are encouraged to put the SIL Standard Format markers at the left margin because it makes proofing and editing easier. But sometimes they don't. It would be a simple matter to write a change table that put each "\" on a new line. It would be as follows:

```
"\" > nl "\"
```

That would work fine, but what if the "\" was part of something else that we wanted to match (like a "\w," for instance)? The "\" has already gone sailing past and been output. The *back(v)* command can help us because it can take characters which have gone "sailing past" the table and put then back into the input file as if they've never come through the table yet.

In the following change table, the first line will catch each "\" that is preceded by an <ENTER> (as SIL Standard Markers ought to be) so that they won't be changed. The second and third lines will catch each "\" that isn't preceded by an <ENTER>, and put an <ENTER> in front of it. There could be a potential problem with the third line, however.

```
nl "\" > dup    c Don't add nl to "\" if not needed!
"\"   > nl "\" back(2)
```

```
"\"    > nl "\" back(2)  c This line is dangerous
```

The danger of the third line is that the program could be caught in a loop. Once the "\" is matched, an <ENTER> is put in front of it, and it is sent through the table again. It would again match at the backslash if the first line of the table had not been included, or if we had said *back(1)* instead of *back(2)*. An <ENTER> would again be put in front of it and it would be sent through the table again... and again... and again...

How can such a thing be prevented? In this case we have included the first line and *back(2)* so there will be no problem with the program getting caught in a loop. This is the most obvious way, turning what is to be backed over into something completely different so that no piece of it will match at the same place again. But that isn't always desirable.

The second way is to be sure to include something to catch the repetitions, such as a switch. (See Section 4.4 [Switches], page 29.) For example, the line could have been:

```
"\" > ifn(checked) set(checked) nl "\" back(2)
      else "\" clear(checked)
      endif
```

It would catch itself when it attempted to send the same backslash through for a second time, thus preventing the loop.

Another solution, sometimes a better one, is to catch the output of that dangerous entry (that might cause the program to go into a loop) and do something else with it. The lines:

```
nl "\" > dup "\"    > nl "***ERROR***" nl "\"
```

can be placed in the table. If a "\" is not at the beginning of a line, the second entry would catch it, and draw attention to it in the output file for later correction, rather than try to fix it, back it into the input file, and match it again.

4.3 Groups

There are two commands associated with the group function. They are:

```
group(name)    use(name)
```

These commands allow certain entries in the table to be available to be matched while others are not.

What *group(name)* Does

This command is not used as part of a "search" > "replace" entry. Rather, it is used on a line all by itself to mark the beginning of a "group" of changes. The changes in this group can be executed as if they were the only changes in the table. The end of the group is marked by either the end of the table, or another *group* command.

What *use(name)* Does

Whenever more than one group is used, each must be appropriately designated by a *group* command. Unless specified otherwise, the program will start in *group(1)*. If numbers are not being used for group names, the program will start in the first group in the table. To begin in some other place, the use command must appear in the *begin* entry. For example:

```
begin > use(2)
```

Although the above example specified *group(2)* as the place to start, any group in the table could have been specified.

The *use(name)* command can also be used to make a different group or set of groups active during processing. The *name* argument tells MSE which group or groups of changes to use from that point on, unless it finds another *use* command. You should put *use(name)* only on the right side of the wedge. Note that when MSE encounters a *use* command, it *finishes the entry* which contains it; *use(name)* does not constitute an exit from the entry.

Example of the *group* and *use* commands

In a bilingual dictionary, one might wish to make certain changes in the orthography of one language without doing anything to the other language. Let's suppose that you had a huge text file that was a bilingual dictionary, one that a Spanish speaker would use to find the meaning of English words. A text file for such a dictionary might be keyed in with each line preceded by a SIL Standard Format marker as follows:

```
\w word in English
\p part of speech in Spanish
\d definition in Spanish
\q qualifying comment in Spanish
\i illustrative sentence in English
\t translation of illustrative sentence in Spanish
```

And suppose the orthography change is to be in the English language. Such a change would affect the *w* and *i* parts of the entry, but not the *p*, *d*, *q*, or *t* parts.

There are two ways, at least, to approach this problem. One way is to use switches. (See Section 4.4 [Switches], page 29.) Another is to use groups. Consider the following table:

```
c Do orthography change for the \w and \i fields
group(1)
'\w ' > dup use(2)   c Go to group two, where the
'\i ' > dup use(2)   c   change occurs

group(2)
'kw' > 'qu'          c Change kw to qu for \w and \i
'\p ' > dup use(1)   c Don't change these fields, go
'\d ' > dup use(1)   c   back to group one, where
'\q ' > dup use(1)   c   nothing happens to kw.
'\t ' > dup use(1)
```

What does this table say? First the line:

```
group(1)
```

identifies the beginning of a group. It tells the program that the following changes belong to a group called *1*. The program will always begin using the changes in group one when the program begins. The lines:

```
'\w ' > dup use(2)   c Go to group two, where the
'\i ' > dup use(2)   c   change occurs
```

tell the program that whenever it sees a `\w` or a `\i` *while it is inside group(1)*, that it should duplicate what it has matched (*dup* command) and go use the changes that are in group two. The next line:

```
group(2)
```

identifies the beginning of the second group of changes. Inside this group, that is, following the *group* command, are the changes, such as:

```
'kw' > 'qu'          c Change kw to qu for \w and \i
```

These are the orthography changes that we want performed on the data in the `\w` and `\i` fields. That is why those markers requested *group(2)*. They are not all that is in *group(2)*, however. The lines:

```
'\p ' > dup use(1)   c Don't change these fields, go
'\d ' > dup use(1)   c   back to group one, where
'\q ' > dup use(1)   c   nothing happens to kw.
'\t ' > dup use(1)
```

catch all the other markers in the dictionary. They send them back to the first group. There, the data following them passes to the output file without any change in "kw", if that combination of letters happens to occur.

4.4 Switches

4.4.1 Introduction

The concept behind using switches is one that is familiar to everyone. The problem is that the concept is not usually formalized. Consider the following statements:

- If it doesn't rain this morning, I'll water the lawn this evening.
- If we have hamburgers this noon, I'll make pork chops for supper; otherwise I'll fix hamburgers.
- If the gate is left open, the dog will run away.
- If we don't get some gas now, we'll run out.
- If George forgets to pick up the groceries on his way home from work, we'll have pork and beans for supper.

Each of these embodies the concept of a switch. If something has or has not happened:

- it rains
- we eat hamburgers
- the gate is left open
- we buy gas
- George remembers the groceries

certain consequences follow or do not follow:

- I water the lawn
- we have pork chops
- the dog runs away
- we run out of gas
- we eat pork and beans

The "something" that leads to the consequences is the condition. Sometimes the consequences follow if the condition occurs:

- If we have hamburgers this noon, I'll make pork chops for supper...

Sometimes the consequences follow if the condition does *not* occur:

- If it doesn't rain this morning, I'll water the lawn this evening.

Of course, *something* happens whether the condition is met or not. If nothing else the consequences fail to occur:

- I don't water the lawn
- The dog doesn't run away
- We don't run out of gas

Sometimes there are alternate consequences:

- I fix hamburgers
- We eat something other than pork and beans

The five statements above can be semi-formalized as follows.

If not (rain in the morning) I will water the lawn this evening.

If (we have hamburgers at noon) we will have pork chops for supper else we will have hamburgers for supper.

If (the gate is left open) the dog will run away.

If not (fill the car with gas) we will run out of gas.

If not (George remembers the groceries) we will have pork and beans for supper.

In each case, the parenthesized condition can be regarded as a switch. Switches have only two states: these are called *on* or *off* (or *true* or *false*; or *set* or *clear*).

A switch by itself doesn't necessarily do anything. The lack of morning rain does not always result in the lawn being watered. I have to decide that circumstances warrant the lawn being watered. Once that decision is made, then I look about for any conditions that would affect my decision: If it rains, I won't need to water lawn. Later, I check that condition (or switch). Did it happen? (Is the switch set?) Then I proceed accordingly.

The nature of a switch, and its particular value, is that it allows something that happened (or didn't happen) in the past to be taken into account for a decision in the present.

For people, remembering the past is no amazing feat; for a computer, remembering the past must be done deliberately. Hence, computers use formal switches. Actually, all of the computer's "memory" is an elaborate array of switches controlled by other switches which are controlled by a program which is a bunch of switches controlled by the data — which sets and clears switches. Fortunately, we need not worry about all these levels upon levels of switch setting and testing. But it is nice to be able to control some levels of it. This allows us to do innovative things with the data.

4.4.2 What the Commands Do

What types of switches are available in the MSE program? How are they used?

There are several commands connected with the switch feature of the MSE program:

```
clear(name)     if(name)
else            ifn(name)
endif           set(name)
```

The *(name)* represents the name of the switch. Other commands may use the same names, but there is no relationship between the names for switches and any other names. Note that the following commands:

```
ifeq(name) 'string'
ifneq(name) 'string'
ifgt(name) 'string'
```

use the same concept of a switch, but the names refer to storage areas, not to switch names. These commands are not described here, but in Section 3.4 [Command Description], page 5. Do not confuse them with the first list of commands, some of which look very similar.

Keep in mind as you read that the terms *set*, *on*, or *true* are used synonymously with one another. The terms *clear*, *off*, or *false* are also used synonymously with one another.

What *set(name)* does

The command *set(name)* causes switch *(name)* to be in the *on* or *true* state. Switch *(name)* will remain set until explicitly cleared. Hence it can be used for reference later as a reminder or signal of what has gone before. The switch is cancelled by the *clear(name)* command. When the table first starts running, all switches are clear, or off.

What *clear(name)* does

The command *clear(name)* causes switch (name) to be in the *off* or *false* state. Switch *(name)* will remain *off* until explicitly set. When the MSE program is started and before any table entries are executed, all switches are cleared, or turned off.

What *if(name)* does

The command *if(name)* checks to see if switch *(name)* is set. If it is set, the commands following the if are executed. If it is not set, the commands following the *if* are not executed (are skipped). This allows commands to be executed only if a certain condition exists.

What *ifn(name)* does

The command *ifn(name)* — which is read "if not *(name)*" — checks to see if switch *(name)* is not set. If it is *not* set, the commands following the *ifn* ARE executed. If the switch *is* set, the commands following the *ifn* are SKIPPED. This allows commands to be executed only if a certain condition does not exist.

What *endif* does

The command *endif* puts a boundary on the *if* or *ifn* command. If the switch was such that the commands following the *if* or *ifn* were being *skipped*, execution of commands will begin again at the *endif* regardless of the setting of any previous switches. (Of course, another *if* or *ifn* may be encountered immediately after the *endif* which would again take into account switch settings.) If the commands following the *if* or *ifn* are being executed, the *endif* has no effect; execution continues.

What *else* does

If switch conditions are such that commands following the most recent *if* or *ifn* are being executed, the *else* command causes the commands following itself to be skipped. If the commands following the *if* or *ifn* are being skipped, the *else* command causes the commands following itself to be executed.

In the following example:

```
"a" > if(test) "a" else "b" endif
```

If switch *test* is set, *a* will go to the output. If switch *test* is not set, *b* will go to the output. The same result could have been achieved by:

```
"a" > if(test) "a" endif
      ifn(test) "b" endif
```

An Example Using Switches

For example, in a bilingual dictionary, one might wish to make certain changes in the orthography of one language without doing anything to the other language. Let's suppose that you had a huge text file that was a bilingual dictionary that a Spanish speaker would use to find the meaning of English words. A text file for such a dictionary might be keyed in with each line preceded by a SIL Standard Format marker as follows:

```
\w word in English
\p part of speech in Spanish
\d definition in Spanish
\q qualifying comment in Spanish
\i illustrative sentence in English
\t translation of illustrative sentence in Spanish
```

And suppose the orthography change is to be in the English language. Such a change would affect the `\w` and `\i` parts of the entry, but not the `\p`, `\d`, `\q`, or `\t` parts.

There are two ways, at least, to approach this problem. One way is to use switches. Another is to use groups (See Section 4.3 [Groups], page 27). Consider the following table:

```
c Do orthography change

'\w ' > dup set(qu)     c Set switch (qu) to change
'\i ' > dup set(qu)     c   kw to qu
'\p ' > dup clear(qu)   c Don't change these fields.
'\d ' > dup clear(qu)   c   Clear switch (qu) so
'\q ' > dup clear(qu)   c   nothing happens to kw.
'\t ' > dup clear(qu)

'kw'  > if(qu) 'qu'     c Change kw to qu for \w and \i
        else dup
        endif
```

What does this table say? The line:

```
'\w ' > dup set(qu)     c Set switch (qu) to change
'\i ' > dup set(qu)     c   kw to qu
```

tell the program that whenever it sees a `\w` or a `\i`, it should duplicate what it has matched (*dup* command) and that it should set switch *qu*. The next lines:

```
'\p ' > dup clear(qu)   c Don't change these fields.
'\d ' > dup clear(qu)   c   Clear switch (qu) so
'\q ' > dup clear(qu)   c   nothing happens to kw.
'\t ' > dup clear(qu)
```

tell the program that whenever it sees any of the other markers, it should duplicate what it has matched and clear switch *qu*, so that switch *qu* will be inactive, in a sense. The next two lines:

```
'kw' > if(qu) 'qu'      c Change kw to qu for \w and \i
       else dup
       endif
```

contain the orthography change we want performed on the data in the `\w` and `\i` fields. They say, whenever a "kw" is encountered anywhere in the data, check to see if switch *qu* has been set. If it has, change it to "qu." Otherwise, duplicate what was matched. Note that this is identical in function to the following:

```
'kw' > if(qu) 'qu'      c Change kw to qu for \w and \i
       endif
       ifn(qu) dup
       endif
```

The *else* command says look at the condition which preceded it. The *endif* command says ignore what preceded, and look at what follows without prejudice.

So a switch in itself is something that can be used to allow the change table to affect data or not affect data, depending on its condition.

4.5 Arithmetic Commands

There are five arithmetic commands:

```
add(name)    mul(name)
div(name)    sub(name)
mod(name)
```

For each command, the syntax is the same:

```
add(name) 'number'
```

These commands are always used on the right side of the wedge. They all work with *numeric strings*. A numeric string is a string that MSE can convert to a value. Valid characters in a numeric string are "0" through "9" . It is valid to precede a string of numeric characters with "-" or "+" . MSE will convert a *numeric string* to a *numeric value*. Such values must be in the range of −2,147,483,639 to +2,147,483,639 (without the commas).

MSE assumes that the specified storage area contains a valid numeric string and that the string following the command is also a valid numeric string. MSE will first convert both the string *in the specified storage area* and the string *following the command* into numeric values. The operation (*add*, *divide*, etc.) is then performed using the two numeric values and the result is stored as a numeric string in the specified storage area, *replacing the store's original contents*.

The *cont(name)* function can be used instead of a numeric string (for more information on the *cont(name)* command, See Section 3.4 [Command Description], page 5). The numeric string following the arithmetic command can be any combination of literal strings and ASCII characters. In other words, the expression '34' is identical to the expression '3' d52 (See Section 3.1 [Form of Changes], page 3 for an explanation of using ASCII characters).

The numeric string following the command is terminated by the next command. In other words, MSE will regard everything following the command up until the next command as part of the string to be operated upon.

There are examples of these commands in Section 3.4 [Command Description], page 5.

5 Comparison with SIL CC

The MSE program is very like to SIL (Summer Institute of Linguistics, Dallas, USA, `http://www.sil.org`) program CC (Consistent Changes) and inspired by it. C++ code of MSE is completely independent from the code of CC (author of MSE never seen it). Differences between MSE and CC version 7.5 are follows, MSE:

* doesn't support *caseless* command;
* capital and small letters differ only in the string literals;
* always use binary mode during changes;
* can use new commands *backi*, *decr*, *not*, *prevsym*, *preci*, *symdup*;
* commands *fol*, *prec*, and *wd* always work good;
* has no limitations on quantity of *prec*, *fol*, and *wd* commands;
* has no limitations on quantity of macros and groups;
* has no limitations on the length of lines in input files;
* has no limitations on number values for *back*, *omit*, and *fwd* commands;
* can use %-comments;
* has no limitations on the depth of *define* command nesting;
* can break long line into pieces by the backslash in the end of each (except last) line-piece;
* can be used as filter. For example:

```
(mse -t - -o - in.txt <<END
'a' > 'b'
'ab' > 'ac'
END
)| wc;
```

* output can be send to the file with the same name as file with input data. For example:

```
mse -t my.mse -o data.txt data.txt;
```

* can generate HTML file-report;
* can correct handle any of 0 (null), 10 (line feed, <ENTER>), 13 (carriage return), 26 (end of file mark in CP/M and MS-DOS) ASCII codes;
* has no debugger;
* has no command line dialog possibilities;
* search side *begin* command may be in any line of changes table;
* has no necessity of using *endfile* or *dup* after *endfile* in search side;
* *ifeq*, *ifneq*, *ifgt* commands always use byte-by-byte string comparison comparing their length first;
* groups order has no any importance;
* *endfile* and *nl* commands is the real commands, not names for the appropriate codes;
* *incr* cannot increase store area size when overflow occurs;
* *define* can use only one argument;
* command line options may be entered in any order;
* has no *-i*, *?*, *-a*, *-q*, *-n*, *-w*, *-m*, and *-s* command line options;
* can use *-h*, *--help*, *-V*, and *-l* command line options;
* change table execution always starts from group *1*;
* can by *excl* command close all active groups;
* one group commands can be broken into pieces, interlaced with contents of other groups.
* has no limitations on capacity of the stores;
* *begin*, *define*, and *endfile* search side commands are global. They belong to all groups.
* all undefined by *define* macro procedures regarded as empty (do nothing) operations.

6 Quick Reference

6.1 Error Messages

This section lists error messages that result from some problem in your table.

All error messages from the MSE program are preceded by `mse:` . Then the error message will be given. If an error is found in the table, the number of line containing the error must be displayed. Use `-l` option if you want to trace MSE computations.

The rest of this section is a list of the errors that MSE generates (number preceding message is the code returned to the operating system):

2 — can't open *filename* for read

Non-existent or incorrect *filename* was entered after `-t` option or as input filename.

3 — can't open *filename* for write

Incorrect *filename* was entered after `-o` or `-l` options.

4 — only one change table allowed

Several `-t` options was encountered in command line.

5 — unknown parameter found

Only `t`, `o`, `l`, `h`, `V`, and `-help` are allowed after `-` in command line.

6 — only one log file allowed

Several `-l` options was encountered in command line.

7 — only one output file allowed

Several `-o` options was encountered in command line.

8 — error in octal number, line *number*

An octal ASCII code was specified, but contains some character other than 0, 1, 2, 3, 4, 5, 6, or 7. In other words, some string of characters has been encountered which is not enclosed in quotation marks that is neither a recognized command nor a valid octal ASCII number.

9 — error in number, line *number*

Some string of characters has been encountered is neither a recognized command nor a valid ASCII number.

10 — error in decimal number, line *number*

A decimal ASCII code was specified, but contains some character other than 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Decimal codes are specified by preceding them with the character *d* or *D*.

11 — error in hexadecimal number, line *number*

A hexadecimal ASCII code was specified, but contains some character other than 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, or F. Hexadecimal codes are specified by preceding them with the character *x* or *X*.

12 — PREVSYM argument less than 1 in line *number*

14 — unknown or misplaced reserved word encountered in line *number*

A string of characters has been encountered which is not enclosed in quotation marks but which also is not a legal command or a legal number. Various control codes will produce this message, since they are not considered legal characters. Only <TAB> and <ENTER> are legal control codes in a table.

Commands which are valid only on one side of the wedge will cause this error if they are used on the other side of the wedge.

This error also produced if after *ifgt(name)*, *ifeq(name)*, or *ifneq(name)* placed command but not *cont(name)*.

15 — unexpected end of line encountered in line *number*

A ' or " has been found in a line without a corresponding ' or " before the end of the line. If several quoted strings occur in the line, any one of them may be missing the quotation mark — the program will not notice that something is missing until the last quotation mark is unpaired.

16 — number too big, must be within the range 0–255, line *number*

Since numbers are used for the 256 ASCII codes, they should be within the range 0 to 255.

17 — duplicated > in line *number*

18 — syntax error in line *number*

After completing literal or command comma or parenthesis have encountered.

19 — literal exceeds *number* bytes in line *number*

No more than 2048 bytes are allowed in one string in the change table. (Entering such a string really possible with many break-line backslashes.) If longer string required break it into parts with *begin end* command sequence.

20 — empty argument in line *number*

21 — ELSE without IF in line *number*

22 — ENDIF without IF in line *number*

23 — END without BEGIN in line *number*

24 — REPEAT without BEGIN in line *number*

25 — duplicated ELSE in line *number*

26 — duplicated ENDIF in line *number*

27 — error in DEFINE statement, line *number*

define is not alone on the search side.

28 — second argument is missing or bad

It may occur in the command with two arguments (such as *ifeq* or *add*) when the first argument is followed by a comma, a parenthesis or a wedge.

29 — duplicated BEGIN in line *number*

Only one initializing entry (with *begin* on the search side) is allowed for a change table.

30 — BEGIN is not alone in search entry in line *number*

31 — duplicated ENDFILE in line *number*

Only one entry with *endfile* on the search side is allowed for a change table.

32 — ENDFILE is not alone in search entry in line *number*

33 — misplaced UNSORTED statement, line *number*

unsorted is allowed only in the initializing entry with *begin* on the search side.

34 — missing open parenthesis, line *number*

The opening parenthesis is missing from a command which expects an argument in parentheses.

35 — missing close parenthesis, line *number*

The closing parenthesis is missing from a command which expects an argument in parentheses.

36 — bad numerical argument in line *number*

One of the *back*, *backi*, *fwd*, *omit*, *prevsym*, or *symdup* commands has been used and the value of the specified argument is non-numerical, negative, or greater than 2,147,483,639.

40 — BACK exceeds buffer

Attempt to get byte from empty output.

42 — INCR buffer is too big

If the specified by *incr* storage area consists of a huge number of the symbols "9" then sometimes it may produce this error. (May be sure that number must be really big, something like 10 in power 10000.)

43 — DECR buffer is too big

If the specified by *decr* storage area consists of a huge number of the symbols "0" then sometimes it may produce this error.

45 — NEXT in the last line of change table

47 — insufficient memory

Current operation ask for dynamic memory and operating system answer that required amount of memory is not available.

48 — arithmetic operation with empty store

Attempt to increment or decrement empty store.

49 — arithmetic operation with non-number store

One of the arithmetic commands (*add*, *sub*, *mul*, *div*, or *mod*) has been used and the contents of the specified storage area does not have a number in it. To resolve this error, you should be sure that the storage area contains only the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, preceded by an optional − or + sign.

50 — arithmetic overflow

One of the arithmetic commands (*add*, *sub*, *mul*, *div*, or *mod*) has been used and either the contents of the specified storage area or the 'numeric string' is outside the range −2,147,483,639 to +2,147,483,639 (do not use commas).

51 — division by zero

Attempt to divide by 0 in *div* or *mod* commands.

54 — can't open temporary file

Operating system cannot create required file.

55 — can't write to temporary file

File system is full or damaged.

56 — SYMDUP exceeds buffer

Attempt to duplicate a character outside the pattern space (match string).

57 — can't seek position in temporary file

Internal error or crash of operating system.

58 — can't read from temporary file

Internal error or crash of operating system.

59 — OutStream::operator[] index too big

Internal error.

60 — InStream::operator[] index too big

Internal error.

6.2 Alphabetical Summary of Commands

%	comment
''	null match; null replacement
add(name)	'number' add numeric string to area *name*
any(name)	match any element of storage area *name*
append(name)	store in area *name*, keep previous contents
back(v)	put last *v* chars output back into input
backi(v)	take last *v* chars from the input stream history and put them into input
begin	beginning of table or nested block
c	comment
cont(name)	match or compare contents of area *name*
clear(name)	clear switch *name*
decr(name)	decrement number in storage area *name*
define(name)	defines a set of commands called *name*
div(name) 'number'	divide value in *name* by numeric string
do(name)	execute set of commands called *name*
dup	duplicate match string
else	else
end	end nested block
endfile	match or output end of job signal
endif	end *if* (applies to all *if*s)
endstore	end storing
excl(name)	exclude (make inactive) group *name*
fol(name)	if following character is in area *name*
fwd(v)	forward *v* characters (does not process)
group(name)	specifies a group called *name*
if(name)	if switch *name* is set
ifeq(name) 'string'	if contents of area *name* equal string
ifgt(name) 'string'	if contents of area *name* exceed string
ifn(name)	if switch *name* is not set
ifneq(name) 'string'	negative of *ifeq(name) 'string'*
incl(name)	include (activate) group *name*
incr(name)	increment number in storage area *name*
mod(name) 'number'	remainder when value in *name* is divided by numeric string
mul(name) 'number'	multiply value in *name* by numeric string
next	perform commands in next entry
nl	match or output new line
not(name)	logical negate switch *name*
omit(v)	omit next *v* characters from input
out(name)	output storage area *name*
outs(name)	output area *name* (storing continues)
prec(name)	if preceding character is in area *name*
preci(name)	if preceding character in the input stream history is in area *name*
prevsym(v)	if *v*-th byte is equal to current

read	read input from keyboard
repeat	repeat from preceding *begin*
set(name)	set switch *name*
store(name)	store in area *name* (discard previous contents)
sub(name) 'number'	subtract numeric string from value in *name*
symdup(v)	duplicate *v*-th byte of match string
use(name)	use group called *name*
unsorted	set using of unsorted change table
wd(name)	if chars before and after in area *name*
write 'string'	output following string to screen
wrstore(name)	output storage area *name* to screen

6.3 Commands by Logical groupings

Commands Using Switches:

clear(name)	clear switch *name*
if(name)	if switch *name* is set
ifn(name)	if switch *name* is not set
not(name)	logical negate switch *name*
set(name)	set switch *name*

Commands Using Store Names or Related to Storage Areas:

add(name)	'number' add numeric string to area *name*
any(name)	match any element of storage area *name*
append(name)	store in area *name*, keep previous contents
cont(name)	match or compare contents of area *name*
decr(name)	decrement number in storage area *name*
div(name) 'number'	divide value in *name* by numeric string
endstore	end storing
fol(name)	if following character is in area *name*
ifeq(name) 'string'	if contents of area *name* equal string
ifgt(name) 'string'	if contents of area *name* exceed string
ifneq(name) 'string'	negative of *ifeq(name) 'string'*
incr(name)	increment number in storage area *name*
mod(name) 'number'	remainder when value in *name* is divided by numeric string
mul(name) 'number'	multiply value in *name* by numeric string
out(name)	output storage area *name*
outs(name)	output area *name* (storing continues)
prec(name)	if preceding character is in area *name*
preci(name)	if preceding character in the input stream history is in area *name*
store(name)	store in area *name* (discard previous contents)
sub(name) 'number'	subtract numeric string from value in *name*
wd(name)	if chars before and after in area *name*
wrstore(name)	output storage area *name* to screen

Arithmetic Commands:

add(name)	'number' add numeric string to area *name*
decr(name)	decrement number in storage area *name*
div(name) 'number'	divide value in *name* by numeric string

incr(name)	increment number in storage area *name*
mod(name) 'number'	remainder when value in *name* is divided by numeric string
mul(name) 'number'	multiply value in *name* by numeric string
sub(name) 'number'	subtract numeric string from value in *name*

Ordinary Number Commands:

back(v)	put last *v* chars output back into input
backi(v)	take last *v* chars from the input stream history and put them into input
fwd(v)	forward *v* characters (does not process)
omit(v)	omit next *v* characters from input
prevsym(v)	if *v*-th byte is equal to current
symdup(v)	duplicate *v*-th byte of match string

Commands Using Group Names:

excl(name)	exclude (make inactive) group *name*
group(name)	specifies a group called *name*
incl(name)	include (activate) group *name*
use(name)	use group called *name*

Commands that can Cause a Loop:

back(v)	if not outputting something different or using a different group
backi(v)	if not outputting something different or using a different group
repeat	if no way to stop repeating
'' (null match)	if not used with *fwd*, *omit*, or *use*
do(name)	
define(name)	if 2 or more defined procedures call each other

Commands Using Defined Procedures (Macros):

define(name)	defines a set of commands called *name*
do(name)	execute set of commands called *name*
next	perform commands in next entry

Commands Involving the Screen or Keyboard:

read	read input from keyboard
write 'string'	output following string to screen
wrstore(name)	output storage area *name* to screen

Nested Block Commands:

begin	beginning of nested block
else	else
end	end nested block
endif	end of conditional set of commands
if(name)	if switch *name* is set
ifeq(name) 'string'	if contents of area *name* equal string
ifgt(name) 'string'	if contents of area *name* exceed string
ifn(name)	if switch *name* is not set
ifneq(name) 'string'	negative of *ifeq(name) 'string'*

repeat	repeat from preceding *begin*

Commands which occur Only on the Search Side:

any(name)	match any element of storage area *name*
define(name)	defines a set of commands called *name*
fol(name)	if following character is in area *name*
prec(name)	if preceding character is in area *name*
preci(name)	if preceding character in the input stream history is in area *name*
prevsym(v)	if *v*-th byte is equal to current
wd(name)	if chars before and after in area *name*

Commands which may occur on Either Side:

begin	beginning of table or nested block
cont(name)	match or compare contents of area *name*
endfile	match or output end of job signal
nl	match or output new line
''	null match; null replacement

6.4 ASCII Codes

ASCII Control Codes and Space Code

Decimal	Hexadecimal	Octal	Character	Abbrev	Meaning
0	0	0	^@	NUL	null
1	1	1	^A	SOH	
2	2	2	^B	STX	
3	3	3	^C	EXT	exit
4	4	4	^D	EOT	end of tape
5	5	5	^E	ENQ	
6	6	6	^F	ACK	
7	7	7	^G	BEL	bell
8	8	10	^H	BS	back space
9	9	11	^I	HT	horizontal tab
10	A	12	^J	LF	line feed
11	B	13	^K	VT	vertical tab
12	C	14	^L	FF	form feed
13	D	15	^M	CR	carriage return
14	E	16	^N	SO	
15	F	17	^O	SI	
16	10	20	^P	SLE	
17	11	21	^Q	DC1	
18	12	22	^R	DC2	
19	13	23	^S	DC3	
20	14	24	^T	DC4	
21	15	25	^U	NAK	
22	16	26	^V	SYN	
23	17	27	^W	ETB	
24	18	30	^X	CAN	
25	19	31	^Y	EM	
26	1A	32	^Z	EOF	end of file
27	1B	33	^[	ESC	escape
28	1C	34	^\	FS	
29	1D	35	^]	GS	
30	1E	36	^^	RS	
31	1F	37	^	US	
32	20	40		SPACE	
127	7F	177		DELETE	

Other ASCII Codes

Decimal	Hexadecimal	Octal	Character
33	21	41	!
34	22	42	"
35	23	43	|
36	24	44	$
37	25	45	%
38	26	46	&
39	27	47	’
40	28	50	(
41	29	51	)
42	2A	52	*
43	2B	53	+
44	2C	54	,
45	2D	55	-
46	2E	56	.
47	2F	57	/
48	30	60	0
49	31	61	1
50	32	62	2
51	33	63	3
52	34	64	4
53	35	65	5
54	36	66	6
55	37	67	7
56	38	70	8
57	39	71	9
58	3A	72	:
59	3B	73	;
60	3C	74	<
61	3D	75	=
62	3E	76	>
63	3F	77	?
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4A	112	J
75	4B	113	K
76	4C	114	L
77	4D	115	M
78	4E	116	N
79	4F	117	O
80	50	120	P

Other ASCII Codes (continued)

Decimal	Hexadecimal	Octal	Character
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5A	132	Z
91	5B	133	[
92	5C	134	\
93	5D	135	]
94	5E	136	^
95	5F	137	_
96	60	140	`
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k
108	6C	154	l
109	6D	155	m
110	6E	156	n
111	6F	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t
117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x
121	79	171	y
122	7A	172	z
123	7B	173	{
124	7C	174	|
125	7D	175	}
126	7E	176	~

7 Literature

1. “Consistent Changes User’s Guide” JAARS, 1991.
2. K. Seitz “Consistent Changes for Publishing” SIL, 1996.