Consistent Changes
User's Guide

Version 7.4
June 1991

Table of Contents

Chapter 1 Introduction To Consistent Changes . . . 3
1.1 Notes on This Manual . . . 3
1.1.1 Purpose . . . 3
1.1.2 Prerequisites for Understanding This Manual . . . 3
1.1.3 Documentation Conventions . . . 3
1.2 How Can Consistent Changes Help Me? . . . 3

Chapter 2 Creating And Using A Change Table . . . 5
2.1 Creating a Change Table . . . 5
2.2 Using a Change Table . . . 5

Chapter 3 Consistent Changes Description . . . 8
3.1 Form of Changes . . . 8
3.2 How Changes are Processed . . . 9
3.3 Order of Changes . . . 9
3.4 Command Description . . . 10
3.5 I/O Options . . . 26
3.6 Running CC from the Command Line . . . 28

Chapter 4 Advanced Features . . . 30
4.1 Storage Commands . . . 30
4.2 The Back Command . . . 36
4.3 Groups . . . 38
4.4 Switches . . . 40
4.4.1 Introduction . . . 40
4.4.2 What the Commands Do . . . 41
4.5 Arithmetic Commands . . . 44

Chapter 5 Quick Reference . . . 46
5.1 Error Messages . . . 46
5.2 Alphabetical Summary of Commands . . . 52
5.3 Commands by Logical Groupings . . . 54
5.4 ASCII Codes . . . 56

Chapter 1 Introduction To Consistent Changes

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 Consistent Changes (CC) program. The purpose of the manual is to fully describe the CC program (CC.EXE), which is part of the Direct Translator Support package. This manual does not include a tutorial for beginners, but a separate tutorial entitled "Using the CC (Consistent Changes) Processor" has been distributed with this "Consistent Changes User's Guide" as a part of the DTS documentation.

1.1.2 Prerequisites for Understanding This Manual

1. Familiarity with the computer to be used and a working knowledge of DOS, including how to change directories and start programs from the DOS prompt.
2. Ability to use ED 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 CC commands and their arguments, reference to a specific part of a CC 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. Also used for names of keys on the keyboard, for example <CTRL>, <ENTER>.
<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 CONTROL 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 Consistent Changes Help Me?

The CC 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.

CC 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, CC can also be used to count words in a text, insert or remove text, or reorder parts of a text.

Chapter 2 Creating And Using A Change Table

2.1 Creating a Change Table

In order to use the Consistent Changes program, you must have a text file that describes the changes you want made. This file is called a change table. There are several change tables in the DTS package for handling scripture. 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 Consistent Changes program as described in section 2.2.

A change table can be created or modified using ED or almost any other word processor. If you are using some program other than ED, be sure that you save your document as an unformatted text file (with line breaks in MS-Word). The filename extension ".CCT" is commonly used when naming CC 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 CC 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 long time, and typing errors might occur in the process.

The following simple CC 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 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 CC program works: CC "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 CC 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 CC program reads the next character.

See the tutorial "Using the CC (Consistent Changes) Processor" for more examples and further explanation of how CC works.

2.2 Using a Change Table

This section assumes that you have found the file CC.EXE on your DTS diskettes, and that it

has been copied into your current directory or into a directory that has been included in your DOS path.

When you run CC, it will ask you for three filenames: the changes file, the output file, and the input file, in that order. The changes file contains the instructions that tell CC what to change. The output file is the file CC will create as it applies the changes to the input file. The input file contains the text you want changed.

The CC 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.

In the following instructions, what you type is in bold print. After typing in the answer to each question, press the <ENTER> key.

At the DOS prompt, type: `CC<ENTER>`

The Consistent Changes program will ask: `Changes file?`

Type the name of the change table file you have created, then press <ENTER>. If you do not include a file extension, the CC program will first look for the file without an extension. If it cannot find a file without an extension, it will look for the file with a ".CCT" extension. If it still does not find the change table, CC will respond:

```
filename not found.  Changes file?
```

You will then have an opportunity to enter the correct change table filename.

The program will then ask: `Output file?`

At this point, type the name of the "after" or output text file, to be created by CC, then press <ENTER>. If you only want to view the output on the screen, you may type **CON:** instead of a file name. If you want to send the output to a parallel printer, type **PRN:** or **LPT1:**. To output to a serial printer, type **COM1:** or **COM2:**. (Users of older versions should note that CC no longer makes use of printer definition files. This means you may not specify something like **P321:** or **EPSONLQ**:. CC will only output to disk files or to physical devices such as **CON:** and **PRN:**.)

When writing the output to a file, if there is already a file by that name in that directory, the CC program will respond:

```
filename already exists.  Replace it [no]?
```

If you wish to keep the existing file with that name (or if you aren't sure), respond **no** <ENTER>, and CC will reprompt for the name of the output file. If you want to overwrite the existing file, answer **y** <ENTER>. (**CAUTION:** If you accidentally typed the name of your input file here, respond **no**; otherwise it will be destroyed.)

The next question CC will ask is: `Input file?`

Type the name of the file containing the text you want changed, the "before" text file. If the program can't find a file you named, it will give you a chance to enter a different input file name. The program's response will be:

```
filename not found.
Input file?
```

After you specify the input file, the program will process your file. Just before it finishes, it will beep and ask:

```
Next input file (<RETURN> if no more)?
```

If you want multiple files combined into one output file, type in the next filename. Your previous file will be completed and the next file will be treated as a continuation of it. Any number of files can be combined in this manner. If you press <ENTER> only, the remainder of your file will be output and the program will then stop, returning you to the system prompt.

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.

If you type **/b** <ENTER> in response to the "Output file?" or "Input file?" questions, the program will back up and ask the previous question again. This allows you a chance to give a new answer to the question if it was answered incorrectly. If a mistake is discovered after pressing <ENTER> in response to the "Input file?" question, your only recourse is press <CTRL> + **c**, then rerun the program.

(See section 3.5 for a description of other I/O options available).

Chapter 3 Consistent Changes Description

3.1 Form of Changes

A change file must be created before CC 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.

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.

3. ASCII codes

It is sometimes necessary to use non-printing ASCII codes in a CC 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 5.4. 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 1-255 are legal before and after the wedge. Note, however, that decimal codes *d10* (linefeed), *d13* (carriage return), and *d26* (end of file) may yield unexpected results and should not be used.

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 + <BACKSPACE>). If you do this, however, *be sure that each ASCII code is expressed using 2 digits* (eg. tilde + <BACKSPACE> should be represented by *x7E08*, not *x7E8*). The hexadecimal codes 1-FF are legal before and after the wedge. Note, however, that *x0A*, *x0D*, and *x1A* may yield unexpected results and should not be used.

Note: *Octal* may be used by *not* preceding the ASCII code with anything (eg. 10 is <BACKSPACE>). The octal codes 1-377 are legal before and after the wedge. Note, however, that 12, 15, and 32 may yield unexpected results and should not be used.

WARNING! Although use of octal numbers greater than 377 will result in an error message, use of octal numbers greater than 100000 will *not* produce an error message.

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 CC 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)* command.

3.3 Order of Changes

Change entries are sorted by CC 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 CC 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 for a more detailed information on *fol(name)*, *prec(name)*, and *any(name)*).

In general, the following rule is used when CC is sorting entries: *any(name)* has the same

weight as one full character, *fol(name)* and *prec(name)* each have weight of 1/10th 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 CC 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, CC 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, CC has an 'unsorted' option so that you can completely suppress CC'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 CC 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: We 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 CC operates.

3.4 Command Description

All commands must be entered in *lower case*.

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 three such commands:

```
back(v)
fwd(v)
omit(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). These strings are case-sensitive. In other words, CC will treat *store(NAME)* and *store(name)* as two different stores.

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*). 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 *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 wordlist program (WDL) 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 over is 300 characters, or to the beginning of the storage area (or output), whichever is less. 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. Note that the number *v* cannot be larger than 127, but more than one *back* command can be used to back up a total of 300 characters.

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.

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, and it must be the first entry in the table. (*Comments*, however, may occur before the *begin* entry). The replacement which follows the *begin* command will be executed before any input data is read. It will not be executed again.

The *endfile* command cannot be used after the *begin* statement for initializing a table. CC will give the error message

```
'CC-F-Defaulting to non-existent group 1'.
```

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' nl  c causes an error
```

This produces an error, because CC considers the *nl* command as part of the string that should be added. 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 "appleWe 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, *surrounded by 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* 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.

caseless — CASELESS MATCH

This command is placed on the right side of the wedge in the *begin* section of the table. (See the *begin* command). It will be applied to all of the input data. The caseless command causes the first letter of a potential match string from the input to be treated as if it were a lowercase letter during the matching process, regardless of whether it was upper or lowercase in the input file. For this reason, the first character in the match string must be lowercase for a match to EVER occur when using caseless. Note: All other characters in the match string are matched exactly, and case is NOT ignored.

If the replacement string begins with a lowercase letter, the case of the first letter of the matched string will be preserved in the output. If the replacement string begins with an up-

percase letter, the first letter of the output string will always be output as an uppercase letter, regardless of the case of the first letter of the input string that was matched.

Note that caseless only works with the alphabetic letters a-z and A-Z. It will not apply to the first character of a string which does not begin with a letter.

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*."

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. You may define up to 127 such sets, distinguishing them by using a different string for *name*. 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. CC 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)* com-

mand 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, up to a "depth" of 10. 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 CC to execute the commands following the *define(1)* command. Those commands happen to include a *do(2)* command which cause CC to execute the commands following the *define(2)* command. Those instructions happen to include a *do(3)* command, which causes CC 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, CC will go back and finish the commands (if any) in *define(2)*. When finished doing *define(2)*, CC will go back and finish the commands (if any) in *define(1)*. When finished doing *define(1)*, CC 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 CC 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. The last element of the replacement *must* also be *endfile* (or *dup*), or the program will not stop. For example:

```
  endfile > out(3) endfile   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 re-routes 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 CC 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,mac,windows)
```

will have the same effect as this table:

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

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, between the match string and the wedge. 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. Although *v* cannot be greater than 127, as many as 300 characters may be forwarded in sequence by using more than one *fwd(v)* command; attempting to move more than 300 characters will be regarded as an error condition, and an appropriate message will be given.

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. You may define up to 127 groups, distinguishing them by using a different string for each *name*. 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:

```
begin > caseless

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

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

CC will begin processing text using the set of changes found in *group(1st)* not in *group(main)* as you might expect. This feature of CC 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, *nl*s, 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. Note that *nl* is considered a character and will not terminate an *ifeq* or *write* command, nor will *c*, *endfile*, or *'' (null)*.

The conditional execution is terminated by an *endif* or *else* command, or by the end of the table entry. Note that when doing comparisons of stores with *ifeq(), ifneq()* and *ifgt()*, any leading zeros that may be in the store will be irrelevant because CC attempts a numerical comparison of stores before it 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 numbers, the actual values are compared. Leading zeros are disregarded. Thus *0011 is greater than 2*. When comparing characters, "greater than" is strictly according to ASCII codes. See the ASCII table at the end of this manual for details. Thus, *b is greater than abc* and *a is greater than B or 1*. CC attempts a numerical comparison first, before it does a byte-for-byte ASCII comparison.

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 CC 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)*. Note that the specified group is

appended to *the end* of the list of groups that CC is using.

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.

Whenever CC tries to increment a character that doesn't exist, it will create the character "0" and then increment the "0" to "1."

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 CC final result would have been "10."

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) endfile   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)*. Note that when doing comparisons of stores with *ifeq(), ifneq()* and *ifgt()*, the leading zeros will be irrelevant because CC attempts a numerical comparison of stores before it does a byte-for-byte ASCII comparison.

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.

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, CC discards the 3 and stores the 7 in storage area *test*. If there is no remainder, CC 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 and a comma. A comma is used as a separator for multiple designators. Letters are case-sensitive. In other words, CC will treat *store(NAME)* and *store(name)* as two different stores. 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 that this function is considered a character sequence (not a command) by such commands as *ifeq*.

Note: You cannot put an <ENTER> between quotes. The only way to indicate an <ENTER> is to use *nl*.

'' — 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. If the table uses a *fwd* or *omit* command on null match, then there should be a separate entry to look for *endfile*. For example:

```
'a'     > 'a'
endfile > endfile  c protect against null
                   c   match at end of file
''      > 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. Although *v* cannot be greater than 127, as many as 300 characters may be omitted in sequence by using more than one *omit(v)* command. Attempting to omit more than 300 characters will be regarded as an error condition, and an appropriate message will be given.

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 any one of the characters contained in the specified storage area. This command can only be used on the search side, between the match string and the wedge. 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:

```
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 (up to a maximum of 10) 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)*, and *any(name)*.

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. CC 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 > caseless
        store(char) ' abcdefghijklmnopqrstuvwxyz,.?!'
        store(count) '0' endstore

any(char) > dup incr(count)
nl > ifgt(count) '59'
        begin
          '**** ERROR count 60 or greater ***' nl
        end
      else
        begin
        incr(count)     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 CC 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. You may use up to 127 switches, distinguishing them by using different names. 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 re-routes the output to an internal storage area. It can only be used on the right side of the wedge. You may have up to 127 storage areas, distinguishing them by using different names. 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)
```

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 in that order. Although up to 127 groups can exist in a table, you can use no more than 25 of them at one time. The groups will be searched in the order they are specified in the *use* command. 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, just *before the wedge*.

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
          endfile
```

3.5 I/O Options

When you type **CC** at the DOS prompt, CC will ask for the name of your:

```
Changes file?
```

After you provide the name of your change table and press <ENTER>, CC will ask for the name of your:

```
Output file?
```

It is in response to these questions that these I/O options may be used:

/b BACKUP TO PREVIOUS QUESTION

This command, in response to any question other than "Next input," will cause the program to re-ask the previous question.

/c COMPILE TABLE

This option will cause CC to compile the table rather than to run the table. It is used after your filename in response to the question, "Changes file?"

```
Changes file? mytable.cct/c
```

If you use the /c option, CC will not ask you for an input file name or an output file name. It will instead ask you for the name it should give to your compiled table that it is about to create:

```
Compiled table file?
```

A compiled table is very compact, usually only a few blocks. This can save table loading time for frequently run tables. The recommended file extension for a compiled table is *.CCC*.

/d DEBUG

This option gives the number of changes and characters in the table, and gives a display of text before and after the changes.

NOTE: This mode will not display properly unless the screen driver, ANSI.SYS is invoked in your CONFIG.SYS at boot up.

The debug option is used after the filename in response to the question, "Changes file?"

```
Changes file? mytable.cct/d
```

The debug option shows the content of the current storage area, any switches on, and the numbers of the current groups.

When you use the debug option you will see a listing of all the stores, switches, and groups used in your change table, before the request for the output file. You will see both a number and a name for each. The number is significant for when you are running CC without the debug option, because all error messages will reference items by number rather than name. The only way to find which name in your table is referenced by a number is to run CC with the debug option.

After you have entered your output and input filenames CC will begin running in the debug mode. CC will stop at the first match and show you the following:

Store *name* **contains: [**contents of current store before match**]**

A line of text showing 35 characters of the "output" text and 35 characters of the "input" text with the matched characters in reverse video

A line of text showing the data after the entire right side of the match is completed

Active groups: *name, name, name . . .* (active groups after match is completed)

Switches set: *name, name, name . . .* (switches that are set after match is completed)

Store *name* **contains: [**contents of current store after the match is completed**]**

A store is shown only if it is currently being stored into. Other stores not shown may also contain data.

In the display of the store contents and the match line, a *nl* character will be displayed as character that looks like a backwards F. The symbol for "end of file" will be a solid block. Other control characters in the data can mess up the screen display.

If more characters are in a store than will fit on one line of the screen, an automatic line wrap is performed.

A null match can be recognized because *no* characters are in reverse video in the match line.

Switches and groups are shown only once because they cannot change between the end of one entry and the beginning of the next. Store contents are shown twice because they can change between matches.

There is no way to display what happens within a replacement, except by seeing the evidence of before and after.

The debug feature of CC starts up in a 'single-step' mode. Simply press a key to go on to the next step. If you want to run steps continuously, press <*ESCAPE*>. Pressing any key will stop the debugger and resume single step mode.

/o NEW OUTPUT FILE

If a /o is typed *instead of* a file name at the "Next Input" question, the program reprompts for new input and output files after finishing and closing the file currently being output. The same change table will remain in effect, but will be restarted from the *begin* statement, as if it had just been loaded. This allows running various files through the same change table without reloading the table every time.

/r RERUN PROGRAM

If a /r is typed *instead of* a file name at the "Next Input" question, the program will return to the beginning of the program, after finishing and closing the file being output. This allows the user to use various change tables without having to return to the DOS prompt.

/t WAIT FOR INPUT

You should never need to use this option. It is outdated and documented here only for the sake of completeness. If a /t is typed after the file name at the "Next Input" question, the program will type the message:

```
Waiting for filename:  Type <RETURN> to continue?
```

and wait for an <ENTER> *before* looking up the file.

/w WAIT FOR SYSTEM

You should never need to use this option. It is outdated and documented here only for the sake of completeness. If a /w is typed after the table name at the "Changes file" question, the program will indicate when the disk may be removed and when to restore it.

3.6 Running CC from the Command Line

When you type **CC** at the DOS prompt, CC asks you a series of questions, such as Changes file?, Output file?, Input file?, etc. You can avoid having to answer all these questions by including your answers when you type **CC**. Note that the following two examples will provide the same results:

```
------------------------------------------------------------
C:\>cc
Consistent Changes 7.4, 15-May-90 Copyright 1987-1990 SIL
Inc.
Changes file? test.cct
Output file? test.out
Input file? test.txt
Next input file (<RETURN> if no more)? <ENTER>

C:\>
------------------------------------------------------------
C:\>cc -t test.cct -o test.out test.txt
Consistent Changes 7.4 15-May-90 Copyright
1987-1990 SIL Inc.

C:\>
------------------------------------------------------------
```

In the second example, the *-t* indicates that the table name will follow and *-o* indicates that the output file name will follow. These must be typed in this order, the same order CC would ask you for the file names if you were not working from the command line. In place of the input file name you can use a *-i* to indicate a file containing a list of input file names. There will only be one output file, however. See I/O options for more information on commands that can be entered along with the file names.

The only thing to remember is that the output file name is preceded by a *-o*, the change table is preceded by a *-t*, and the input file name doesn't get preceded by anything. In fact, you don't even have to remember this! If you type **CC**? at the DOS prompt, CC will display a list of which "-" goes with which file name!

Summary of CC Command Line Options

-t Change Table name; if *compiling*, this is the name of the *uncompiled* file; if *running* a change table, it can be either an *uncompiled or compiled* change table

-o Output file or device

-i Name of file containing a list of input files

This file can be created with the SIL Editor or any other word processor provided the output is unformatted (plain ASCII). In the list each file name is followed by a <RETURN> as in this example:

```
matthew.scr
mark.scr
luke.scr
john.scr
acts.scr
```

-m Ask "Next input" question; when running from the *command line*. The default is to *not ask* the question when running from the *command line*. When running with *CC prompts*, the program will ask for "Next input" after processing each file.

-s Compiled table name; used only when compiling

Chapter 4 Advanced Features

4.1 Storage Commands

There are five commands directly connected with the storage feature of the CC 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)          ifgt(name)
any(name)          incl(name)
cont(name)         incr(name)
div(name)          mul(name)
excl(name)         prec(name)
fol(name)          sub(name)
ifeq(name)         wd(name)
ifneq(name)        wrstore(name)
```

More information on these can be found in section 3.4

The expression *(name)* represents any logical name you choose. In older versions of CC only numbers could be used to identify stores and groups, etc. The logical name feature has been included since version 7.2B. 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. The names are case sensitive, so *store(cat)* and *store(Cat)* would refer to different stores. There is a program limit of 127 different stores. 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 temporay 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 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 CC 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)
            endfile         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 Consistent Change 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 file:"

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

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. Notice that after the entry is output, there is another *endfile* command. 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 DOS prompt. If we don't send it back out, the output file will never be closed and the program will never end!

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)
          endfile          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 a part 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.

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 the irregular 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 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 Standard Format 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 CC). People are encouraged to put the 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 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.) 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)*, (or in the first group whose name *begins* with a '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 CC 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 CC 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 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.) Another is to use groups. Consider the following table:

```
c Do orthography change for the \w and \i fields

begin > caseless
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)*. Unless it is told otherwise, 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" or "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 CC program? How are they used?

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

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

The (name) represents the name of the switch. There can be up to 127 of these in a single table. 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 *name*s refer to *storage areas*, not to switch names. These commands are not described here, but in section 3.4. 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 CC 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 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). Consider the following table:

```
c Do orthography change
begin > caseless

'\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? First the line:

```
begin > caseless
```

tells the program to ignore case when it matches. For this table, it means that "kw" and "Kw" will be changed with the same match. Unless it is told otherwise, the program will always consider case when it matches. The lines:

```
'\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)
- div(name)
- mod(name)
- mul(name)
- sub(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 CC 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 "+". CC will convert a *numeric string* to a *numeric value*. Such *values* must be in the range of -1,999,999,999 to +1,999,999,999 (without the commas).

CC assumes that the specified storage area contains a valid numeric string and that the string following the command is also a valid numeric string. CC 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). 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 for an explanation of using ASCII characters).

The numeric string following the command is terminated by the next command. In other words, CC will regard everything following the command up until the next command as part of the string to be operated upon. Note that *nl* is considered a character and will not terminate an arithmetic command, nor will *c*, *endfile*, or ' ' (null).

There are examples of these commands in section 3.4.

Chapter 5 Quick Reference

5.1 Error Messages

This section lists error messages that result from some problem in your table. It does not list error message that may result from misspelling input file names or table names when starting CC from the DOS prompt.

All error messages from the CC program are preceded by *?CC-E-*, *?CC-F-*, *?CC-W-*, or *CC-Warning-*. If an error is found in the table, the line containing the error may be displayed, with an arrow pointing at the place at which CC realized the error. Then the error message will be given. Error messages will specify particular stores, groups, switches and defines by numbers rather than names. You may have called a store *junk*, but any error associated with it will display a number instead. You can find out which store is referenced by that number by running CC with the debug option.

Some errors can be caused by others. For example, an error in a group name will cause all references to that group to also appear to the CC program to be errors.

If there are errors in your table, the following will be printed after the table has been processed:

```
There were errors in the change table.
Correct the errors and rerun.
```

The rest of this section is a list of the errors that CC generates:

Arithmetic: divide by zero in group name

The *div(name) 'number'* command has been used and the *'numeric string'* has a value of 0. To resolve this error, you should change *'number'* to be something other than zero.

Arithmetic: non-number in group name

One of the arithmetic commands (*add(name) 'number', div(name) 'number', mod(name) 'number', mul(name) 'number',* or *sub(name) 'number'*) 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.

Arithmetic: number greater than 2,000,000,000 in group name

One of the arithmetic commands (*add(name) 'number', div(name) 'number', mod(name) 'number', mul(name) 'number',* or *sub(name) 'number'*) has been used and either the contents of the specified storage area or the *'numeric string'* is outside the range -1,999,999,999 to + 1,999,999,999 (do not use commas).

Arithmetic: overflow in group name

One of the arithmetic commands (*add(name) 'number', div(name) 'number', mod(name) 'number', mul(name) 'number',* or *sub(name) 'number'*) has been used and has resulted in a number outside the range -1,999,999,999 to + 1,999,999,999 (do not use commas).

Backed too far storing

The number of *back(v)* commands given exceeded the number of characters in the current storage area.

Backed up too far

More than 300 characters were backed using the *back(v)* command.

Bad number

The value *v*, which should be a number, is not a number or is missing altogether. Commands which expect a numerical argument are *back(v)*, *fwd(v)*, and *omit(v)*.

Begin command not first in table

A *begin* was found on the search side of an entry, but it was not the first entry in the table.

Binary command not in begin section

This error results from improper use of an incompletely implemented feature. This error occurs when the *binary* command is used in your table other than in the *begin* section of your table.

Caseless command not in begin section

This error occurs when the *caseless* command is used in your table other than in the *begin* section of your table.

Decimal number too big, must be less than 256.

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

Defaulting to empty group x

This error occurs when there are no change entries! To resolve this error, add either a change entry or an *endfile* section to your table.

Do nested deeper than 10

Nesting of defined sets of commands is allowed, that is, *define(name)* commands are allowed to call other *define(name)* commands using a *do(name)*, and those can call yet other such commands. However, there is a limit how deep such nesting can go. That limit is 10. Check that there is not a loop, or that a *define* command does not call itself. Both these error conditions are illustrated below:

```
Loop:   define(1)  ... do(2) ...
        define(2)  ... do(3) ...
        define(3)  ... do(1) ...

Calling Itself:  define(1) ... do(1) ...
```

Do(name) used but never defined

This error results when a *do(name)* statement is encountered but there is no corresponding *define(name)* statement in the table.

FATAL ERROR! excl command in group (name) removes all active groups

An *excl(name)* should only be used if you have previously made more than one group active with multiple *use(name)* commands, and only then if you leave at least one group active.

Font section of table is ignored in CC

You may get this error if you try to run a table that has no changes listed in it! More commonly, it is because the table you are trying to run contains a *font(name)* command, which is not a valid CC command. *Font(name)* used to be a valid CC command, but is no longer. The *font(name)* command needs to be removed before the table will run as expected.

Fwd too many

A maximum of 300 characters may be forwarded with the *fwd(v)* command, even if they were only moved one character at a time.

Group (name) multiply defined

This error means that identical *group(name)* commands exist in your table in more than one place. If you have several *group(name)* commands in your table, you must have a different number or name in place of *name* in each one.

Keep in mind that entries which appear before the first *group(name)* command are considered to form a default *group(1)*. If a subsequent explicit *group(1)* appears in the table, CC will abort with a "Group 1 multiply defined" message.

Group (name) excluded but not active

An *excl(name)* command was used to exclude a group that was not active. *Excl(name)* can only be used on groups that have been activated with *use(name)* or *incl(name)*.

Group command not in front of change

A *group(name)* command has been encountered, but the next line does not contain a wedge. Comments may follow a *group* command, but check that they are preceded by <ENTER> *c* <SPACE> or <SPACE> *c* <SPACE>.

Illegal command following arithmetic operator

A command that performs an arithmetic operation (such as *add(name) 'number'*, *div(name) 'number'*, *mod(name) 'number'*, *mul(name) 'number'*, or *sub(name) 'number'*) has been used without the *'number'*.

Illegal command following comparison operator

A command that performs a comparison (such as *ifeq(name) 'string'*, *ifgt(name) 'string'*, or *ifneq(name) 'string'*) has been used without the *'string'*.

Illegal 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.

Illegal parenthesis

A parenthesis was encountered where no parenthesis is legal. This could be an extra parenthesis —*set(1))* or *ifeq(1)(3)*— or a parenthesis for a command which does not take an argument in parentheses. The following are commands which do not take an argument in parentheses:

```
begin      end         nl
c          endif       ''
caseless   endfile     read
dup        endstore    repeat
else       next        write --use wrstore(name)
```

Illegal use of command after >

The following commands cannot be used on the replacement side of an entry:

```
any(name)   define(name)   fol(name)   group(name)
prec(name)   wd(name)
```

Illegal use of command before >

Only the following commands are legal on the search side of an entry:

```
any(name)   cont(name)     endfile   nl   prec(name)
begin    define(name)   fol(name)    ''   wd(name)
```

The *group(name)* command must *not* be followed by a wedge on the same line.

Invalid decimal digit

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*.

Invalid hexadecimal digit

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, or F. Hexadecimal codes are specified by preceding them with the character *x* or *X*.

Invalid number for arithmetic

One of the arithmetic commands (*add(name) 'number', div(name) 'number', mod(name) 'number', mul(name) 'number',* or *sub(name) 'number'*) has been used and the *'numeric string'* is not a number. To resolve this error, you should be sure that the *'numeric string'* contains only the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, preceded by an optional - or + sign.

Line too long, end cut off

A line in the table exceeded 125 characters.

Missing parenthesis

The opening parenthesis is missing from a command which expects an argument in parentheses *(name)*.

More than 10 prec()s in succession

No more than 10 *prec(name)* commands can be used together.

No close parenthesis

Either the closing parenthesis is missing from a command which expects an argument in parentheses or there is an invalid character in the argument.

No wedge on 'begin' line

There is a *begin* command that is not followed by a wedge on the same line.

No definition for do

A *do(name)* command has been used, but the corresponding *define(name)* command was never made.

Number cannot be zero

Zero was used as the numerical argument of a command. Zero is not a legal value.

Number too big

A number given as the numerical argument for either a *back(v), fwd(v),* or *omit(v)* command exceeds 127. The maximum value allowed for *v* is 127. A maximum of 300 charac-

ters may be backed, forwarded, or omitted by using more than one command in a row. (eg. *back(120) back(30)* causes 150 *back*s to occur.)

Number too big, must be less than 2,000,000,000
One of the arithmetic commands (*add(name) 'number', div(name) 'number', mod(name) 'number', mul(name) 'number',* or *sub(name) 'number'*) has been used and either the contents of the specified storage area or the *'numeric string'* is outside the range -1,999,999,999 to +1,999,999,999 (do not use commas).

Number too large
An octal ASCII number exceeds 377 octal.

Omit too many
More than 300 *omit* characters were done in CC.

Storage overflow of store (n)
So many characters have been stored that the storage area is overflowing. When CC is running, there is room in memory for approximately 61,000 characters to be stored. This room is shared between all the storage areas. It is also used by other things in your table. This error occurs when storage is being done but there is no room for any more characters.

Store name used but never stored into
One of the following commands has been used on a storage area that has not first been opened with a *store(name)* command:

```
ifeq(name)   ifneq(name)   outs(name)
ifgt(name)   out(name)     wrstore(name)
```

Switch name tested but never set or cleared
Either a *if(name)* or *ifn(name)* command has been used on a switch that has not first been set with a *set(name)* command or cleared with a *clear(name)* command.

Table too large.
The total volume of the table exceeds the capacity of the program to store it. (This does not include comments, which are discarded during the storage process.) The maximum size table that CC can handle is a compiled 64K table. See section 3.5 for information on compiling a table.

Too many changes.
No more than 2500 changes can be made in a CC table.

Too many defines
A table cannot have more than 127 different *define(name)* commands.

Too many groups
A table cannot have more than 127 different *group(name)* commands.

Too many stores
A table cannot have more than 127 different storage areas. These storage areas are created with the *store(name)* and *append(name)* commands.

Too many switches
A table cannot have more than 127 different switches. These switches are created with the *set(name)* and *clear(name)* commands.

Unmatched quote

A ' or " has been found in a entry without a corresponding ' or " before the end of the entry. If several quoted strings occur in the entry, 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.

Unrecognized keyword

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 ASCII octal 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.

Use of more than 25 groups

More than 25 groups cannot be in active use at once in CC. Although up to 127 groups may exist in a table, only 25 may be made active using the *use(name)* and *incl(name)* commands.

Use of nonexistent group

A *use(name)* command was specified in the table, but the corresponding *group(name)* does not occur.

Use(name) encountered, but group never defined

A *use(name)*, *incl(name)*, or *excl(name)* command has specified a group that does not exist in the table.

WARNING

Error messages which begin with this word are bringing something to your attention of which you might not be aware. CC will probably be able to use your table anyway. The message following the WARNING can be looked up in this section.

Width must be right after wedge

The *wid(name)* command was used in a font entry, but is not immediately after the wedge. Only spaces and tabs may precede the *wid(name)* command before the wedge. This message only occurs when a /m command has been given to compile a table for Manuscripter.

5.2 Alphabetical Summary of Commands

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
begin	beginning of table or nested block
c	comment
caseless	ignore case of first character of match
cont(name)	match or compare contents of area *name*
clear(name)	clear switch *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 file char
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
''	null match; null replacement
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*
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*
use(name)	use group called *name*
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
/b	backup to previous question
/c	compile table
/d	debug trace
/m	compile table for MS

/o	new output file
/r	return to beginning
/t	change input media
/w	wait for system

5.3 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
set(name)	set switch *name*

The following are similar to if(name) in function, but use *store* names, not switch names:

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'*

Commands Using Store Numbers or Related to Storage Areas:

add(name) 'number'	add numeric string to storage 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*
div(name) 'number'	divide storage area *name* by numeric string
endstore	end storing
fol(name)	if following character is in area *name*
ifeq(name)	if contents of area *name* equal string
ifgt(name)	if contents of area *name* exceed string
ifneq(name)	negative of *ifeq(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 area *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*
store(name)	store in area *name* (discard previous contents)
sub(name) 'number'	subtract numeric string from storage area *name*
wd(name)	if chars before and after in storage area *name*
wrstore(name)	output storage area *name* to screen

Arithmetic Commands:

add(name) 'number'	add numeric string to storage area *name*
div(name) 'number'	divide storage area *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 area *name* by numeric string
sub(name) 'number'	subtract numeric string from storage area *name*

Ordinary Number Commands:

back(v)	put last *v* chars output back into input
fwd(v)	forward *v* characters (does not process)
omit(v)	omit next *v* characters from input

Commands Using Group Numbers:

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
repeat	if no way to stop repeating
‘ ’ (null match)	if not used with *fwd*, *omit*, or *use*
endfile	if matched and not output on right
define(name)/do(name)	if 2 or more defined procedures call each other

Commands Using Defined Procedures:

define(name)	defines a set of commands called *name*
do(name)	execute set of commands called *name*
next	do next set of replacement commands
caseless	process data in “caseless mode”

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

Commands which occur Only on the Search Side:

any(name)	match any element of storage area *name*
fol(name)	if following character is in area *name*
prec(name)	if preceding character is in area *name*
wd(name)	if chars before and after in area *name*
define(name)	defines a set of commands

Commands which may occur on Either Side
(Note that the usage of these commands may be different on opposite sides of the wedge):

begin	beginning of table or nested block
cont(name)	match or compare contents of area *name*
endfile	match or output end of file char
nl	match or output new line
‘ ’	null match; null replacement

5.4 ASCII Codes

ASCII Control Codes

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	

Other ASCII Codes

Decimal	Hexadecimal	Octal	Character
32	20	40	SPACE
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	~
127	7F	177	DELETE