Consistent Changes for Publishing

PAD-CCP

written by
Karelin Seitz

edited by
Kenneth Hubel

International Publishing Services
Summer Institute of Linguistics
7500 W. Camp Wisdom Rd.
Dallas, TX 75236
(972) 708-7440
FAX: (972) 708-7388

8/1/96

8/1/96

Table of Contents

Table of Figures

Introduction

The Consistent Changes program (CC.EXE) is a powerful program with many uses. This course is designed specifically for those applications involved in preparing a manuscript for publishing. All examples and exercises are taken from that context. Course coverage includes all CC commands and many helpful techniques.

Constructing a CC table is very much like computer programming. Therefore, those individuals with programming experience or aptitude will probably advance more quickly and perhaps further in their understanding of CC than those without such background/aptitude. However, *this should not discourage anyone from this course!* There are several different levels on which one may work with CC. Within the publishing context, much of the work with CC is in using established CC tables which may require only minor job-specific modification.

Each individual has his own strengths, weaknesses, and aptitudes. It has often been found that those having the 'programming' aptitude for CC lack some of the other aptitudes for camera-ready page production—such as aptitudes for graphic layout or for repetitive work requiring thoroughness and attention to detail. There is a need for a mix of individuals with varying talents and aptitudes to be involved in different capacities in the publishing process. But it is important for all of them to have some level of understanding of Consistent Changes.

This course is designed to allow each individual to work at that level which is appropriate for him. The following course objectives are stated in terms of three different levels of proficiency. Each student should set his expectations according to his background, aptitudes, and the requirements of his specific assignment.

Course Objectives

Proficiency Level 1 (On the job, the individual will need direct supervision by someone with in depth understanding of CC and the publishing process.)

General: The student will be able to modify an existing CC table at well-defined designated points and run CC using the modified table.

Specific: The student will be able to activate designated options in the VPNAM.CCT change table and run CC using this modified table.

Proficiency Level 2 (On the job, the individual will be able to work semi-independently with occasional assistance from someone with in depth CC knowledge.)

General: The student will be able to trace the logic flow through limited sections of an existing complex table, will be able to make simple to moderate modifications in an existing complex table, will be able to write simple to moderately complex tables, and will be able to run CC from prompts or command line.

Specific: The student will be able to make appropriate job-specific modifications to VPNAM.CCT and FLAGEM.CCT, write increasingly complex (simple to moderate) job-specific tables, and run CC from a command line entry.

Proficiency Level 3 (On the job, the individual will be able to work independently, handling any level of complexity.)

General: The student will be able to understand and extensively modify existing complex tables, will be able to write and debug increasingly complex tables, and will be able to run CC using any of the operation options.

Specific: The student will be able to appropriately modify FLAGEM.CCT or FIXEM.CCT and write and debug a table of equal complexity.

Course Exercises

The exercises at the end of each module in this course can be divided into three categories that relate to the proficiency levels outlined in the Course Objectives. A student's ability to successfully complete the exercises within each group can give him a general idea of his degree of understanding of the Consistent Changes program.

The exercises in the first category involve basic concepts of CC that most students will be able to grasp, even with little programming or computer aptitude. Successful completion of these exercises would be *preliminary* to attaining Proficiency Level 1:

Module	Exercise	Module	Exercise	Module	Exercise
3	1	9	2	14	1
4	1	9	3	14	2

The exercises in the next category involve simple modifications of existing change tables and the creation of single entry tables. Successful completion of these exercises would indicate the attainment of Proficiency Level 1 of the Course Objectives:

Module	Exercise	Module	Exercise	Module	Exercise
2	1	5	2	10	2
2	2	5	3	11	1
3	2	6	1	12	1
3	3	6	2	12	2
4	2	7	1	15	2
5	1	8	3	15	3

The exercises in the last category involve more complex modifications of existing change tables and the creation of tables that require greater logical skills. Successful completion of these exercises would indicate the attainment of Proficiency Level 2 of the Course Objectives:

Module	Exercise	Module	Exercise	Module	Exercise
4	3	8	2	13	1
7	2	9	1	13	2
8	1	10	1	15	1

If a student finds that he is able to successfully complete all exercises in the course with little or no consultation and without feeling overwhelmed, then he is well on his way to attaining Proficiency Level 3 of the Course Objectives.

Orientation for TUTORED Self-Paced Use of Course

This course is to be used in a *learning environment* with an *accessible* tutor/trainer! It has not been designed for solo use. It is for *self-paced* learning, but not *self-taught* learning.

Required materials include this syllabus, the Consistent Changes User's Guide, Consistent Changes software (CC.EXE version 7.4), and the student disk for this course.

It is recommended that the student work through each module—completing all activities, questions, reading and exercises, ensuring that the module objectives have been met, and reviewing work with the tutor—before moving on to the next module. Modules build on each other. Once material has been covered, it is assumed that the information is understood, and it will be used in later modules without further explanation.

People learn in different ways. Some need to read the material first or see it illustrated. Some need to interact with others, discussing the information, or *hearing* it presented. Others prefer 'hands on' experimentation. The course is designed to accommodate all learning styles. If you know how you learn best, emphasize that approach, but do not ignore the other avenues for learning. There are unique advantages within each.

Orientation for Classroom Use of Course

This course is also well suited to a classroom environment of 4-10 students, with the trainer giving comprehensive instruction to the students before they work out the exercise problems.

The tentative classroom schedule is:

Week 1		Week 2	
Monday	Mod 1, 2, 3	Monday	Mod 10, 11
Tuesday	Mod 4, 5	Tuesday	Mod 12, 13
Wednesday	Mod 6, 7	Wednesday	Mod 14, 15
Thursday	Mod 8, 9	Thursday	Review, Mod 16
Friday	Review	Friday	Review, Finish up

To best utilize classroom instruction, it is recommended that the student *preview* the modules that will be taught prior to coming to class; an *understanding* of the modules is not necessary at this time. During the morning classroom session the student will receive comprehensive instruction covering those modules. In the afternoon the student should review all material covered in class, then work out the assigned exercises to gain a further understanding of the concepts taught.

The following chart summarizes when the sections of each module should be covered:

	students preview before class	trainer covers in detail during class	students review and do exercises after class
Commands/Topics Covered	X		
Objectives	X		
Instruction	X	X	X
Vocabulary and Concepts	X	X	X
Practice Activities and Questions		X	X
Reading Assignment			X
Exercises			X

Typographic Conventions Used in This Syllabus

The typographic conventions used in this syllabus will hopefully become self-evident while you read the text. A summary of the primary conventions used are as follows:

SMALL.CAP	Names of files, most of which are located on the student disk
`Typewriter`	CC commands and keyboarded computer input
<ENTER>	Name of key pressed in a single keystroke
<CTRL/C>	Names of keys pressed concurrently—much in the same way as producing upper case (shifted) letters
text words	Words defined in Vocabulary sections
`data stream`	CC text data stream showing output text, search pointer, and input text
16 endstore	Page number and paragraph name to be read in the CC User's Guide (used in reading assignments)
italic	Emphasized text

Note that the typographic conventions used in the CC User's Guide are slightly different from those shown here for this syllabus. Consult the table in the User's Guide for the typographic conventions used there.

Distribution of Course Materials

This course syllabus and associated student disk is used in the Consistent Changes for Publishing course taught by International Publishing Services (IPub) at the International Linguistics Center in Dallas.

The course materials (syllabus and disk) are available at any time from IPub as a self-paced course with tutor assistance. Additionally, the materials may be purchased from IPub for $25.00 plus postage.

Permission is granted to copy the materials as long as the syllabus and disk files are not altered. Any corrections or suggested changes should be submitted to IPub–User Support, ATTN: PAD-CCP, at the address below.

For those using the materials outside of IPub, a registration fee per student ($10 for SIL member or employee, $20 for non-SIL) will entitle the student to receive a critique of their exercises, and, if satisfactorily completed, IPub Course Completion certificate.

To register for a classroom or self-paced course at IPub, or to purchase the syllabus and disk, or for further information contact:

User Support
International Publishing Services
7500 W. Camp Wisdom Rd.
Dallas, TX 75236 USA

phone: (972) 708-7364
fax: (972) 708-7388

Mod 1 Introduction to Consistent Changes

OBJECTIVES

At the end of this module, the student (from memory) will be able to:

- draw a block diagram of CC showing inputs and output,
- name at least three ways CC is used in preparing a manuscript for publishing.

INSTRUCTION

1. **What is Consistent Changes?**
   Consistent Changes (CC) is a computer program written by JAARS. It will apply a set of specified changes to one or more text files in a consistent manner. In its simplest form it functions like the search and replace feature of a word processor. However, CC can also be used to count specific items, insert or delete items, extract or reorder items, and report on conditions found. It can also do these things in a context sensitive way.

2. **How does Consistent Changes work?**
   CC must have two inputs: the file to be changed and a file describing the changes to be made. CC produces one output file. Neither of the input files are changed by running CC. (See Fig. 1)

   The file describing the changes to be made is called a change table. The file name for the change table is usually given the extension .CCT to designate it as a CC change table. The change table must be created before running CC. Any word processor or text editor that can create an ASCII file may be used.

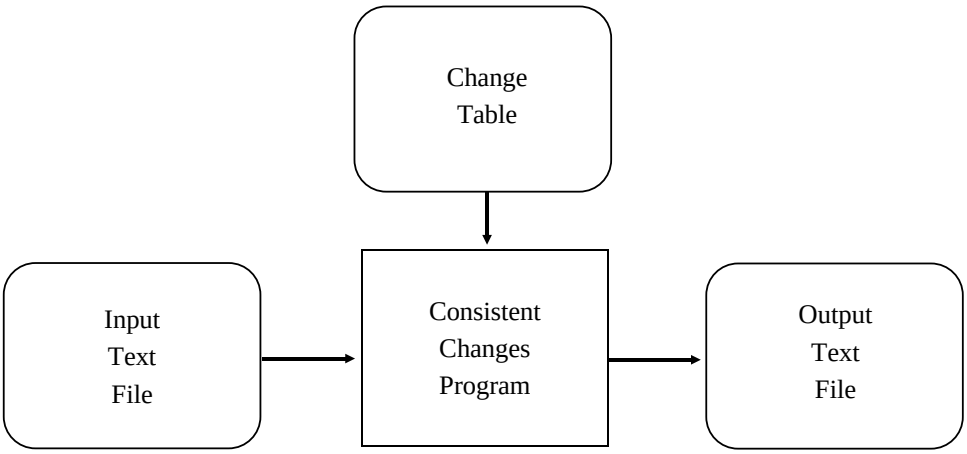


Fig. 1. Input and Output Flow

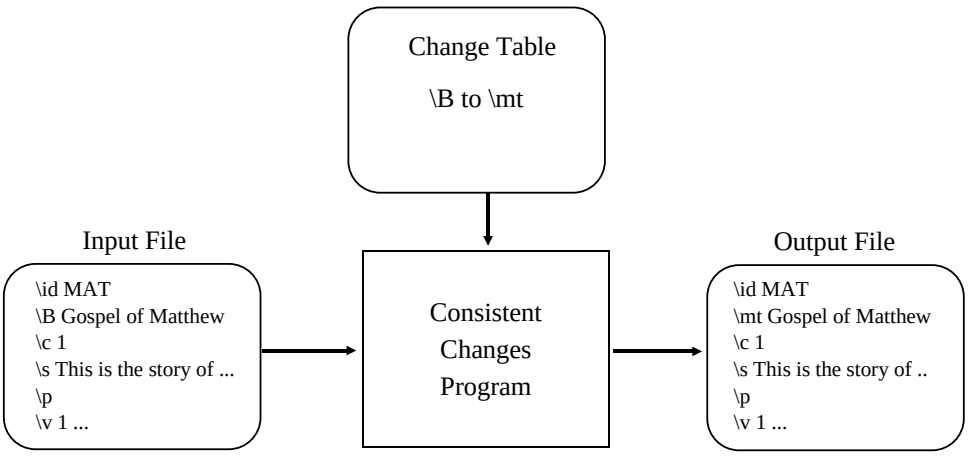


Fig. 2. Input and Output Flow Sample

The basic process flow of CC is summarized in Figure 2. The change table and the input file are read and compared. When input text matches an entry in the table, the corresponding change is made in the output; any input text that is not matched in the change table is passed on to the output unchanged.

As with all computer programs, specific commands and correct syntax are required to describe the changes that are to be made so that the CC program will be able to correctly perform them. This course will teach you how to properly describe changes in the change table to produce the desired changes in the text file.

3. How is Consistent Changes used in preparing manuscripts for publishing?

CC is one of the most powerful and versatile tools we have for identifying, verifying, and correcting generic codes, punctuation, orthographic representations, spelling and other aspects of the manuscript contents and structure.

Some of the specific uses include:

- error flagging
- counting words, elements, etc.
- correcting spelling, Standard Format Markers (SFMs), etc.
- cleaning up unnecessary spaces, markers, etc.
- reordering text elements
- extracting specific text elements for review or separate processing
- stripping unneeded text elements
- exchanging SFMs for format commands used by the formatting program
- inserting text attributes
- exchanging digraphs for a single substitute character or ASCII code
- marking valid potential hyphenation points in the text

VOCABULARY and CONCEPTS

ASCII file—a text file that contains only standard ASCII codes, with no formatting or control codes (e.g., output from JAARS ED program, or MS-Word when file is saved unformatted.)

change table—in Consistent Changes, the table specifies what is to be searched for in the text; under what conditions it will be considered a match; and what action to take when a match occurs.

consistent change—the same change is applied in the same way each time the same specified circumstance is encountered.

context sensitive—both the 'search' and the 'replace' (or action to be taken), can be restricted or changed by the surrounding text or conditions encountered.

digraphs—two or more successive characters used to represent a special character or accented character.

search and replace—there are two components: the 'search' component and the 'replace' component. The text is searched for the 'search' component. When it is found, it is replaced with the 'replace' component.

syntax—the way in which commands or instructions are spelled or structured according to the specific program's requirements.

QUESTIONS

1. The search and replace feature on a word processor can do everything that Consistent Changes can do.

   ☐ T ☐ F

2. The input file will be altered according to the changes specified by the change table when CC is run.

   ☐ T ☐ F

3. Name three ways that CC is used in preparing manuscripts for publication.

   _____

   _____

   _____

READING ASSIGNMENT

CC User's Guide: 5 Creating a Change Table

(Interpret this to mean: Read the section entitled "Creating a Change Table" found on page 5 of the CC User's Guide at the back of this syllabus.)

Mod 2 Basic CC Syntax

COMMANDS/TOPICS COVERED

search & replace	**character string**	**comments**
right wedge	**delimiters**	**running CC**

OBJECTIVES

At the end of this module the student (using notes, syllabus, and/or CC User's Guide) will be able to:

- create and/or modify a change table file to make simple orthographic or Standard Format Markers (SFM) changes.
- document the table internally with comments
- run CC from the DOS prompt

INSTRUCTION

1. **Search and Replace Arguments—** An entry in a change table consists of two basic parts: an item to search for, and an action to take when the item searched for is found. These parts are known as the search argument and the replacement argument, respectively. They are written in the change table with a right wedge between them as follows:

   ```
   "search argument"  >    "replacement argument"
   ```

   A space or tab must precede and follow the wedge. The search argument and wedge must fit on *one* line; but the replacement argument may be multiple lines.

   One thing that can be searched for is a character string. It must be enclosed in delimiters to indicate to CC that this is data rather than an instruction. In CC, either the apostrophe or inch mark (' or ", but not the grave accent) may be used as a delimiter, and the beginning and ending delimiters must match. Strings containing both ' and ″ must be represented in pieces (though together they are still considered a single character string):

   ```
   'The man said, "I '   "don't "   'remember."'
   ```

   A character string can also be used as a replacement argument. A simple entry in a change table would be:

   ```
   "character string 1"     >    "character string 2"
   ```

   This would search for "character string 1" in the input file and,

when found, would replace that string in the output file with "character string 2".

NOTE: Some text processors such as XyWrite require pressing <ENTER> at the ends of every line of the change table to ensure that they terminate with a CR/LF (carriage return and line feed). Do not assume that the editor's "auto-wrap" feature automatically adds the CR/LF simply because it looks right *on the computer screen*. (The JAARS ED program *does* auto-wrap properly.) Change tables keyed using MS-Word must be saved in "text only" mode with line breaks. All lines in a change table must contain 125 characters or less from the beginning of the line to its corresponding CR/LF. Longer lines will be truncated, resulting in potential errors.

2. **Commands—** A command is a word that has special meaning to the CC program. Commands function as instructions to the CC program. All commands must appear in lower case letters. Some commands require a parenthesized argument.

   Commands are never enclosed in delimiters. If delimiters are used, CC will treat the commands as character strings. Commands are always surrounded by space (i.e., a space, a carriage return, or a tab). (Note: if a command requires a parenthesized argument, the space must occur *after the parenthesized argument*, not after the command itself.)

3. **Comments—** Comments can be inserted into the change table to explain what an entry is doing or why it is being done, as well as what the purpose of the table is, who wrote it, when, etc. Comments are an *extremely* important part of any change table that is more than a few lines long or will be used more than once.

   A change table frequently becomes a standard tool requiring only minor modification for each use. Comments are necessary for even experienced CC users to understand and modify change tables.

   A comment consists of the letter c surrounded on both sides by space (i.e., a space, a carriage return, or a tab), with the text following it on the same line. The c may be at the beginning of a line making the whole line a comment, or to the right of an entry making the rest of the line a comment. The letter c may be either lower or upper case. (This is the *only* command that may appear upper case.)

   The following example includes various comments:

```
c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c
c                                                                   c
c  This table makes simple SFM corrections.                       c
c                                                                   c
c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c

       "\B"  >  "\mt"    c  change \B to \mt

c other SFMs to be changed may be added below.
```

4. **Running CC from the DOS prompt—** After you have created a change table file, it is very simple to run the CC program. The program will ask you for the information it needs. To run CC from the DOS prompt, enter:

   ```
   CC                <ENTER>
   ```

   Note: CC.EXE must be in a directory that DOS knows to search. Ordinarily, this means it will be in a directory which is listed in the PATH* command in the AUTOEXEC.BAT* file on your computer. When you run CC, your default directory* should be the directory your change table (and/or input) is in.

   After entering CC, assuming DOS was able to find the program, it will respond with the program version and date, and then will ask:

   ```
   Changes file?
   ```

   Respond with the name of your change table file.

   ```
   Changes file?     mytable.cct     <ENTER>
   ```

   If no extension is entered for the file name, CC will first look for a file by that name without an extension, but if none is found, it will append .CCT to the file name. Remember to add the path name if the change table is not in the default directory.

   CC will then ask:

   ```
   Output file?
   ```

   Note that the output file prompt *precedes* the input file prompt! Respond with the file name you want given to the output file to be created.

* If these terms are not familiar to you, consult your DOS manual.

```
Output file?      myoutput.txt    <ENTER>
```

If a file by that name already exists, the program will notify you with:

```
(filename) already exists, Replace it? [No]
```

The `No` in brackets is the default answer if you just press <ENTER>. `No` or <ENTER> will result in `Output file?` being asked again.

If the file does not already exist or if you answer the message `"(filename) already exists. Replace it? [No]"` with `Y` (for yes), the program will immediately create a file with zero content by that name. For this reason you must *never* name the output file the same as the input. The input file will be zeroed out and you will have *lost it*!

After the output file name is established, the program will ask:

```
Input file?
```

Respond with the name of your input file including the path if it is not in the default directory.

```
Input file? c:\mydir\myinput.txt   <ENTER>
```

When the end of the input file is encountered, the program will ask:

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

This allows you to continue processing additional files and the output to be combined into one output file. Pressing <ENTER> signifies that there are no more files to be processed, and the program will finish up.

VOCABULARY and CONCEPTS

character string—one or more characters representing literal text rather than a command or ASCII code. A string may be used on the search or replacement side and must be enclosed in single or double quote delimiters.

command—a word having specific meaning to the CC program. It is typed in lower case letters and not enclosed in delimiters.

delimiters—a character used at the beginning and end of an item to indicate the item's boundaries. In CC, `'` or `"` can be used as a

delimiter, but the beginning and ending delimiter for an item must be the same. Examples: `'search'` and `"search"`

`LIST`—LIST.COM is a shareware utility that will display a file on the screen. It is for browsing only as it does not have editing capability. It has been included on the student disk for use in this course.

right wedge—the character also known as a closing angle bracket. This character signifies the end of a search argument.

```
c  AFRIORTH.CCT: group 10 orthographic changes for Africa.

group(10)     c ORTHOGRAPHY TABLE

              c     PUNCTUATION

   '(('     > '['      c square open bracket
   '))'     > ']'      c square close bracket
   '*fn*'   > '<P10MJ230>n<P255DJ0>'  c ftnote mrkr in text
   '--'     > '<197>'  c EM dash
   '='      > '<->'    c dup discretionary hyphen
   '<<'     > '<169>'  c English open double quote,
   '>>'     > '<170>'  c English closing double quote
c  '<<'     > '<214>'  c French open dbl quote (alter CST)
c  '>>'     > '<215>'  c French closing dbl quote (alter CST)
   '<'      > "'"      c English open single quote
   '>'      > "'"      c English closing single quote
c  '>'      > "'"      c French closing sgl quote (alter CST)
c  '<'      > "'"      c French open sgl quote (alter CST)

              c     DIACRITICS
   "'a"     > '<141>'  c acute lc a
   "'e"     > '<142>'  c acute lc e
   "'i"     > '<143>'  c acute lc i
   "'o"     > '<144>'  c acute lc o
   "'u"     > '<145>'  c acute lc u
c  "'/e"    > '<146>'  c acute lc epsilon
c  "'/o"    > '<147>'  c acute lc au
c  "'/u"    > '<148>'  c acute lc barred u MUST ADD TO CST
   "'A"     > '<149>'  c ACUTE UC A
   "'E"     > '<150>'  c ACUTE UC E
   "'I"     > '<151>'  c ACUTE UC I
   "'O"     > '<152>'  c ACUTE UC O
   "'U"     > '<153>'  c ACUTE UC U
c  "'/E"    > '<154>'  c ACUTE UC EPSILON
c  "'/O"    > '<155>'  c ACUTE UC AU
c  "'/U"    > '<156>'  c ACUTE UC BARRED U MUST ADD TO CST
```

PRACTICE ACTIVITIES and QUESTIONS

1. The change table (AFRIORTH.CCT) on the preceding page is used for African languages to change digraphs and multi-stroke representations of characters into single characters or ASCII codes.

   Things to notice in this change table:

   – the use of apostrophe and inch marks as delimiters
   – the alignment of the entry components to aid readability
   – the use of space or tabs separating the entry components
   – the use of comments:
     a. at the beginning of the table to state its name and purpose
     b. centered on lines to form headings (e.g., `c DIACRITICS`)
     c. following an entry to identify the resulting character
     d. at the beginning of a line, preceding an entry, to make that entry inoperative but still available for future use

2. In the AFRIORTH.CCT example preceding, draw a circle around the actual characters that will be searched for. (Do not include delimiters.)

3. Using AFRIORTH.CCT, what will the following sentence be changed to in the output file?

```
\v 1 Net'a mat'a ap'i oko ut'o i gamena nene,
'Omas'imini agepag'u gakot'o mina v'emo
nene hanuva minake, mino-loko iteko minam'o.
```

_____

_____

_____

4. Write out the entry, with a descriptive comment, that you would add to AFRIORTH.CCT to change the two-character sequence `"e` to a lower case 'e' with dieresis, or 'ë', represented by the sequence <158>.

_____

5. Write a change table (complete with comments) that will make the following Standard Format Marker (SFM) changes:

   change \b to \mt _____

   change \sh to \s_____

   change \x to \r _____

   change \pp to \p _____

6. Locate the CC.EXE program on your hard disk. At the DOS prompt, type: `disk cc.exe <ENTER>`, or use some other utility. What is the complete path name where it is located?

   _____

7. `LIST autoexec.bat`. Is the path for CC listed in the PATH command in AUTOEXEC.BAT?

   ☐ Yes ☐ No

8. Insert your student disk in drive A: and make that drive the default drive by typing the command `A:` at the DOS prompt. Create a new subdirectory on your disk: `md \output`. Run CC using AFRIORTH.CCT as the change table and AFMT.SFM as input. These are on your student disk. Name the output file AFMT.OUT and cause CC to create it in the new "output" directory on your student disk.

READING ASSIGNMENT

CC User's Guide: 5-7 Using a Change Table, **8** Form of Changes, **13** comment

EXERCISES

1. With the A: drive as your default drive, make a subdirectory on your student disk with your own name: `md (name)`. Now copy and rename AFRIORTH.CCT from the root of your student disk to MYAFRIOR.CCT in *your* directory on the disk by typing:
   `copy \afriorth.cct \(name)\myafrior.cct`.
   Using your copy of MYAFRIOR.CCT, do the following things:
   a. after the first comment line add a comment line stating:
      `Modified for PAD-CCP Mod 2 by (your name) – (date)`
   b. uncomment search entry `'/d'` (the hooked d) and change the replacement to `'#'`;
   c. add an entry to change the sequence space-hyphen-space to `<196>`, including a comment labeling this an en-dash;

d. `list afmt.sfm` and note the changes that will be made;
e. run CC using your modified table and input file AFMT.SFM;
f. `list` your output file and check for changes—especially the `'#'` and `<196>`;
g. print your output, write your name on it, and hand it to your trainer.

2. Write a change table (first on paper and then key it into the computer) containing the following:
   a. comment lines identifying name of table, purpose of table, date written, and author;
   b. documented entries for the following changes:

`\mt` to `@main title =`	(tag for main title)
`\s` to `@sect hd =`	(tag for section head)
`\c` to `@chpt =`	(tag for chapter num)
`\p` to `@par =`	(tag for paragraph)

   (for this exercise, do not include spaces after the SFMs in the search strings, or after the '=' in the replacements.)
   c. run CC using AFMT.SFM as input;
   d. look at the input and output files to check changes;
   e. print your change table, write your name on it, and give it to your trainer.

Mod 3 Setting the Stage Before You Begin

COMMANDS/TOPICS COVERED

search order	**nl**	**caseless**
search pointer	**begin**	

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of the sorted order of entries by identifying correct output when given input and table entries;
- describe to the trainer the basic operation of the search pointer.
- write a table entry correctly using `begin` and `caseless`;
- demonstrate an understanding of `caseless` by correctly filling in the output that will result from given input and table entries.

INSTRUCTION

1. **Input files, output files, and data buffers—** As has already been mentioned, input files which are read into the CC program are not changed. An input file will remain intact, untouched, by the CC program when the program is done. The end product of the program will be a newly created output file, the contents of which are generated according to the instructions in the change table.

   CC accomplishes this by reading the input file and making an identical copy of its contents in a data buffer within the computer memory. A data buffer is simply a place designated in computer memory for retaining data, in this case, the characters read from the input file, while a program is running.

   All changes that are done to the input text will actually be taking place in this buffer. When all the changes are done, then the contents of this data buffer are copied to an output file on the computer disk for permanent storage. In this course, when we speak of input text or output text, we will usually be referring to the characters within the data buffer.

2. **Search order and search pointer—** While constructing a change table that will yield desired results, it is helpful to understand the order in which table entries are searched and how Consistent Changes makes use of a search pointer in the data stream. First, you should know that CC doesn't search the input text for the search arguments in the order that they are entered into the table. CC sorts them by length

with the longest search entry first:

EXAMPLE

Table entries:			Search order:		
"t"	>	"x"	"these"	>	"those"
"the"	>	"a"	"the"	>	"a"
"these"	>	"those"	"t"	>	"x"

Using the above table entries with the following input text:

```
these are times that try the student
```

you will obtain the following output text:

```
those are ximes xhax xry a sxudenx
```

and *not* this text, which would result from *not* sorting the entries:

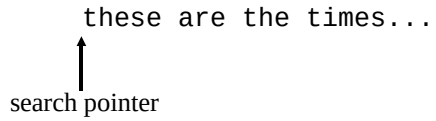
```
xhese are ximes xhax xry xhe sxudenx
```

Now let's look in greater detail at how this search is made in the input text, or more specifically, in the data buffer or *data stream*. CC uses a pointer to track its forward progress through the input text. You might call it a *search pointer*.

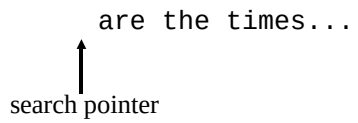
The search pointer tracks the current place in the data stream where all matches and changes take place. Unless otherwise instructed by various CC commands or unless a match occurs, the search pointer moves through the data buffer one character at a time.

The search pointer divides the input text from the output text in the data buffer. All characters to the right of the pointer are input text characters, originally copied from the input text file. They have not yet been compared to the search entries of the change table. All characters to the left of the search pointer are output text characters, already processed by the change table, and ultimately to be copied to the output text file at the completion of the program.

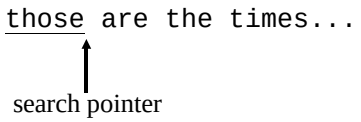
When the CC program begins, the search pointer will be pointing in front of the first character of the input text. The program will be ready to compare the data found to the right of the pointer with each search argument, starting with the longest search entry first, looking for identical matches. Using the above table entries and input text as an example, the following shows the initial data and pointer location:



The input text is compared to the longest table entry `"these"`. This entry exactly matches the first five characters of the data stream and is considered a *match*. When a match occurs, the entire matched string is removed from the input text:



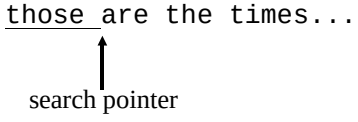
then the replacement string `"those"` is placed in the output, that is, on the left side of the search pointer, becoming:



In this course we will signify the output characters in the data stream by an underline.

With the search pointer in its new position, the characters in the data stream beginning with `" are..."` (remember that each space is a character too!) is once again compared to the sorted list of entries. It is compared with the longest entry `"these"` first, but it does not match. The next longest entry `"the"` does not match, and neither does the final entry `"t"`.

When the characters following the search pointer will not match any search entry, we say "there is no match," and the data stream is handled in a special way. At a "no match" the data stream is treated as though exactly one character is matched and an identical character is placed in the output. In our example, a space is removed from the input and a space is placed in the output:



And as the search continues the computer will:

– compare the data stream (beginning at the search pointer) with the search entries, longest to shortest;

– remove the matched characters from the input;

– place the replacement string in the output (left of the pointer);

– when there is no match, remove one character from the input and place it in the output.

Notice that the search pointer does not point *to* characters but *between* characters, and that text may be added to or deleted from the data stream *only* at the position of the search pointer.

3. `nl` **(new line)—** When searching for a phrase, it is important to remember all the forms it might take. Let's consider phrases that might have line breaks in the middle. In ASCII files, each line ends in a hard return. Therefore,

EXAMPLE

```
"The Gospel According to St. John" > "replacement"
```

would match the input text:

```
The Gospel According to St. John
```

but would not match the input text:

```
The Gospel
According to
St. John
```

with a return at the end of each line.

To match on this input text, we must search for a return or new line in those specific locations using the command `nl`. The following search would match the above input text:

EXAMPLE

```
"The Gospel" nl "According to" nl "St. John" >
                                "replacement"
```

The command `nl` can also be used to force a new line in the output by using it in the replacement argument.

EXAMPLE

```
"The Gospel" nl "According to" nl "St. John"  >
                                "The Gospel According to"
                                nl "St. John"
```

4. `begin`— If there are certain actions you want the Consistent Changes program to do before any of the input file is read, this can

be indicated by the `begin` command. When used, it must be used by itself as the search argument in the *first* entry in the table.

```
begin  >    replacement argument
```

It will only be executed once. (It may be preceded by comments, but it must be the first executable entry in the table.)

5. `caseless`— This command is used only on the replacement side of a `begin` statement. It enables a single search string to match a string in the data stream whose initial character might be either an upper or lower case letter. However, `caseless` works in a unique way—so don't jump to conclusions! Here are some things you need to know about `caseless`:

   a. In `caseless` mode, CC checks the case of the first *(and only first)* character of the input text (i.e., the single character immediately to the right of the search pointer). If it is an upper case letter, it is changed to lower case *before* the input text is compared to the search entries of the change table. Lower case and non-alphabetic characters remain the same.

      The second, third, etc., characters of the input string are not altered. If the second character is upper case, it remains upper case.

      Since the first character of the input text will always be looked at in its lower case form when using a `caseless` table, the first character of each search entry (if alphabetic) must likewise begin with a lower case letter to be matched. Search entries beginning with upper case letters will *never* be matched.

   b. If the first character of the input was originally upper case and changed to lower case before matching, then the first character of the replacement string that is to be sent to the output will be changed to upper case. If the first character of the output is non-alphabetic or already upper case, it will be output as is.

EXAMPLE

Table entries:

```
begin  >  caseless      c line 1
"The"  >  "One"         c line 2
"the"  >  "a"           c line 3
"few"  >  "Some"        c line 4
"(one" >  "(this"       c line 5
"HIS"  >  "YOUR"        c line 6
```

When this is found in Input:	This is the resulting Output:	Comments:
`The`	`A`	matches at line 3 (NOT 2!)
`the`	`a`	matches at line 3
`Few`	`Some`	matches at line 4
`few`	`Some`	matches at line 4 (notice `"S"`)
`(One`	`(One`	will NOT match at line 5!
`HIS`	`HIS`	will NOT match at line 6!

c. Even when a search or replacement string is broken in parts (e.g. `"the Gospel" nl "According to" nl "St. John"`), *only* the first character of the *entire* string is affected by the `caseless` command.

d. A change table will not function partly in regular mode and partly in `caseless` mode. It is either *all* `caseless` or none. In other words, you cannot put `caseless` in a replacement other than the initial `begin` statement in order to suddenly change searching modes.

VOCABULARY and CONCEPTS

hard return—the equivalent of pressing the <ENTER> or <RETURN> key.

pointer—a "place keeper" in the data stream. Don't be concerned with *how* the place is kept, but rather the concept of using a pointer.

PRACTICE ACTIVITIES and QUESTIONS

1. You won't be familiar with most of the commands, but `LIST` the following change tables (found on your student disk) and notice the ways in which `begin`, `caseless`, and `nl` are used.
   a. 7CHAR.CCT—Notice `begin > caseless` is the first executable line of the table. Search for `nl`. See how it is used on the replacement side two times in `group(1)` and on the search side in `group(4)`.
   b. RESPEL.CCT—Notice that `caseless` is not the first replacement action for `begin`, but it is *one* of the replacement actions. Notice the use of `nl` on the search and the replacement sides in `group(1)` and `group(10)`.

READING ASSIGNMENT

CC User's Guide: 9-10 Order of Changes, **12** begin, **13** caseless, **21** nl

EXERCISES

1. On a blank sheet of paper, write the table entry that is needed to allow search entries and input text characters to match whether the first character of the input is capitalized or not. Save this paper for Exercise 2 and 3 below.

---

2. Carefully study the following table entries. For each entry, write down (on the paper from Exercise 1) all input text strings (if any) that can match the search string, and the corresponding output string that will result from each input string.

Table Entries

```
begin        >   caseless
'whosoever'  >   'whoever'      c  entry A
'verily'     >   'truly'        c  entry B
'LORD'       >   'God'          c  entry C
'christ'     >   'Jesus'        c  entry D
'\sH '       >   '\s '          c  entry E
'gOSPEL'     >   'good news'    c  entry F
```

Table entry	All input strings (if any) that will match the search entry	Resulting output string when input string at left is matched
A.	whosoever	whoever
	Whosoever	Whoever
B.		
C.		
D.		
E.		
F.		

3. Given the following change table and line of input, write out the expected output on the same sheet of paper used in Exercise 1 and 2, and turn it in to your trainer.

Table:

```
begin      >  caseless
"a"        >  "x"
"man"      >  "person"
"manage"   >  "handle"
```

Input:

```
A man can manage a manager.
```

Output:

_____

Mod 4 Grouping Table Entries

COMMANDS/TOPICS COVERED

group **incl** **excl**
use

OBJECTIVES

At the end of this module, the student will be able to:

- draw a block diagram showing how control is passed from group to group when given a change table containing three groups;
- modify and/or create a change table containing at least three groups correctly using `group`, `use`, and `incl` or `excl` commands.

INSTRUCTION

1. **Groups—** Consistent Changes allows you to put table entries into groups so you can control which entries are to be used under different conditions. Each group is labeled with the command `group` immediately followed by a unique name or number within parentheses identifying that group. This command is on a line by itself with no wedge or replacement argument:

```
group(1)
```

   Here's some additional information you should know about groups:

   – although names or numbers may be used to identify groups, numbers will make the groups easier to find in a large table;

   – when numbers are used as the group labels, they need not be consecutive but should be in ascending order for readability;

   – a group ends at the start of the next group or at the end of the table;

   – there is a limit of 127 groups.

2. **Use—** Groups can be made active by the `use` command. The `use` command is a replacement argument and is immediately followed by the name or number of the group containing the search entries to be used. The name or number identifying the group is in parentheses.

   When CC begins, `group(1)` will be active unless otherwise specified

EXAMPLE

```
c       WDLSTRIP.CCT  Mod 1     20-JUN-89 (modified)
c       Strips WDL.EXE output of everything
c       except the words.

group(1)  c send text to output until next sfm
   '\'       >  use(2)          c SFM found
   nl '\'    >  use(2)

group(2)  c identify sfm
   'id'      >  '\id NIV 2 John words' use(3)
   'w '      >  nl use(1)       c retain the word
   'n'       >  use(3)          c strip the count
   'l'       >  use(3)          c strip the reference

group(3)  c strip unneeded elements
   '\'       >  use(2)          c new SFM found
   endfile   >  endfile
   ''        >  omit
```

Fig. 3. WDLSTRIP.CCT Change Table

by a `use` command in the `begin` statement. If `group(1)` does not exist, then the group physically encountered first in the table will be active.

Let's discuss what is taking place in Figure 3.

By default, `group(1)` is the active group at the outset. CC operates as if `group(1)` were the only group in the table. `'\'` and `nl '\'` are the only compares made of the input. When a backslash is found, the replacement argument containing `use(2)` is performed. Now `group(1)` is no longer active. `Group(2)` search entries will be the only ones compared against the text until CC is instructed otherwise by another replacement argument. Now CC searches the input for `'id'`, `'w '`, `'n'`, or `'l'`.

If `'id'` is found in the input, a character string is sent to the output, then `group(3)` becomes the active group. If `'w '` is found, a new line (`nl`) is sent to the output and `group(1)` becomes active again. If `'n'` or `'l'` is found, `group(3)` becomes the active group.

What is all of this accomplishing? When you follow the flow of logic through the table, you will find that `\id` will cause a character string to be sent to the output, `\w` will cause the word following it to go to the output, and `\n` or `\l` will cause the characters following them not to go to the output (due to the null match and `omit` command in `group(3)` which we will discuss later). This means that the output will consist of only the \id information and a list of words, each starting on a new line.

EXAMPLE

Input	Output

`\id WDL`
`\n 00001`
`\w acknowledge`
`\l 2JN 0:7`
`\n 00001`
`\w Antichrist`
`\l 2JN 0:7`
`\n 00003`
`\w anyone`
`\l 2JN 0:9,10,11`
`\n 00002`
`\w as`
`\l 2JN 0:6,7`
	`\id NIV 2 John words`
	`acknowledge`
	`Antichrist`
	`anyone`
	`as`

Fig. 4. Sample Input and Output

Figure 4. is a sample of what the input might look like and the resulting output.

If you can understand how our CC table can result in the output shown from the above input, you may skip to point 3. below. If not, let's follow some of this input data through the change table.

Control starts with `group(1)` so we are comparing for a `'\'` or `nl '\'`. We find that `'\'` matches the first character in the input. The replacement side of that entry passes control to `group(2)`. (The backslash that was matched was not replaced by the replacement argument, so it is not sent to the output.)

In `group(2)` we are comparing for `'id'`, `'w '`, `'n'`, or `'l'`. The next two input characters are `id` and match the first `group(2)` entry. The replacement side sends `'\id NIV 2 John words'` to the output and then passes control to `group(3)`.

`Group(3)` compares for the next backslash. If the next input character is not a backslash, then the pointer is moved one character to the right. (The commands used in the second and third lines of `group(3)`, which cause the pointer to be moved, will be discussed in a later module.) The next characters in the input consist of `<space>WDL<newline>\n<space>00001`... Everything up to the `\n` will be dropped, one character at a time—nothing will be sent to the output.

When the backslash is encountered, nothing is sent to the output, but

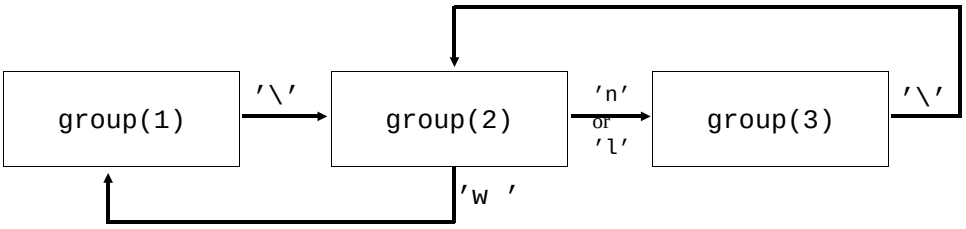


Fig. 5. Block Diagram

control is again passed to `group(2)` to identify which SFM follows. SFMs '`n`' or '`l`' result in control passing to `group(3)` to locate the next backslash with still nothing going to output.

When a `\w` is encountered, the backslash matches in `group(3)`, control goes to `group(2)`, and the `w` matches. Then a `nl` (new line) is sent to output and control goes to `group(1)`. The biggest difference between `group(1)` and `group(3)` is that `group(1)` does not contain an entry like the last entry in `group(3)` so input that is not matched is passed to output.

The block diagram in Figure 5. is one more way of describing the transfer of control from one group to another. There is one block representing each group in the table. The arrows indicate the control passing from group to group. The lines are labeled with the search entry that must be matched for the transfer of control to take place. For instance, when control is in `group(1)`, a backslash must be matched for control to pass to `group(2)`; in `group(2)`, a match on `'w '` will pass control back to `group(1)`.

If you need more help, ask your trainer to step you through the flow. It is important that you understand *now* how the input data causes the control to be passed from one group to another.

3. Other features about groups

a. multiple groups active—More than one group can be active at a time. This can be accomplished by specifying more than one group in a `use` command:

EXAMPLE

```
search argument          >   use(2,10)
```

Notice that the groups are contained within the same set of parentheses and are separated by commas with no spaces in between.

b. search order when using multiple groups—In Mod 3, we learned that search entries were sorted into descending order of length before the compares began. When groups are used, the entries of different groups are not sorted together. Entries are sorted within each separate group. When multiple groups are active, the groups are used in the order in which they are specified in the `use` command.

   In other words, the following table:

EXAMPLE

```
Group(1)
'search argument' >     use(10,2)

Group(2)
 'c'         >  'replacement'    c  entry 1
 'ip'        >  'replacement'    c  entry 2
 'v'         >  'replacement'    c  entry 3
 'q2'        >  'replacement'    c  entry 4

Group(10)
 ':/o'       >  'replacement'    c  entry 1
 '\:_u'      >  'replacement'    c  entry 2
 "'a"        >  'replacement'    c  entry 3
```

   would result in the following order of compares—`group(10)` entries 2, 1, 3; then `group(2)` entries 2, 4, 1, and 3.

c. beginning with groups other than the first group—The initially active group can be changed from the first group to another group (or groups) by specifying the group(s) in the `begin` statement:

EXAMPLE

```
     begin      >  use (1,10)
```

or

```
     begin      >  use (2,5,66)
```

d. currently active groups list—The `use` command should not be thought of as a "go to" command. Processing *does not* proceed to the named group as soon as the `use` command is issued. Rather, the `use` command merely updates a "currently active groups list" which CC will later consult when it is ready to begin a search for another match.

   When two or more `use` commands occur within a replacement, the currently active groups list will be updated each time a `use` is executed, but only the *final* status of the currently active groups

list after the replacement is completely finished will determine which group(s) will be active at the start of the next search.

4. Including and excluding groups

a. `incl`—The `incl` command is similar to the `use` command in that it specifies one or more groups. However, rather than making *only* the specified group(s) active (as in the `use` command), the `incl` command tells CC to *add* the specified group(s) to the *end* of the currently active groups list. This way you don't have to know exactly what groups are active—you just want a specific one or more groups to be added to those that are active.

NOTE: A bug in CC version 7.4 causes a change table to malfunction when an *already-active* group is specified in the `incl` command. It is, however, proper to do this in CC version 7.5! In the newer version, specifying an already-active group in the `incl` command will result in that group being removed from the active groups list, and then placed at the end of the list, thereby making that group to be searched *last* if other groups are active.

b. `excl`—the `excl` command is the opposite of the `incl` command. It ensures that the specified group(s) is(are) not active.

EXAMPLE

```
group(1)
"string A"  >  use(2,10)

group(2)
"string B"  >  excl(10)
"string C"  >  incl(10)
"string D"  >  use(1)

group(10)
"string E"  >  "string X"
```

5. In summary— To trace the steps that CC goes through *each time* the input data is searched for a match:

- **Check active groups list** for which groups (and order) will be used for comparing search strings to the input data;
- **Change to lower case the first character** after the search pointer, if the table is in `caseless` mode;
- **Find an exact match**, testing strings from longest to shortest within each group;
- **Remove the matched string** in its entirety from the input;
- **Do all replacement actions** specified by the match;
- **Change to upper case the first character** of the replacement *if* the first character of the match had been previously changed to lower case.

PRACTICE ACTIVITIES and QUESTIONS

1. By studying CAPCHK.CCT in Figure 6., trace the flow from group to group for the following input text sequences. Don't be concerned about the commands we haven't covered—just look at the `groups` and entries with `use`.

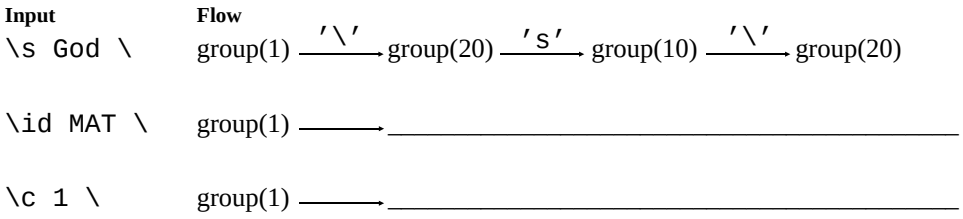


Fig. 6. CAPCHK.CCT

```
C    CAPCHK.CCT                Mod 1   15-SEP-88
C    (Modified for PAD-CCP Mod 4, May 1991, by K. Seitz)
C    Locates sentence initial words not beginning with UC.
C    You must modify store 1,2,3,5,&6 for your data!
C     "   "      "      group 1&20     "     "  "

begin       >
     store(1) 'ABCDEFGHIJLKMNOPQRSTUVWXYZ' endstore  C UC
     store(2) 'abcdefghijlkmnopqrstuvwxyz' endstore C lc
     store(3) '"/' endstore            C diacritics
     store(4) '1234567890' endstore    C numbers
     store(5) ' (<' nl endstore        C sent init punct
     store(6) ' )>' nl endstore        C sent final punct
     use(1)

group(1)
     '\'     > use(20)         C SFM found, check it out.
     '.'     > next
     '?'     > next
     '!'     > set(1) use(10)  C Final punct found,
                               C next should be U.C.
     endfile > clear(1) endfile
     ''      > clear(1) omit   C Strip everything else

group(10)                  C 1st LETTER SHOULD BE UC
     '\'     > use(20)     C SFM was found
     '*f' any(2) '*' > ''      C Footnote
     any(3) > ''               C a diacritic
     any(5) > ''               C opening punctuation
     any(6) > ''               C closing punctuation
     any(1) > ''               C an UC LETTER  (modified for
     any(2) > ''               C a lc letter   this example)
```

(continued on next page)

Fig. 6 CAPCHK.CCT continued

```
group(20)                  C PROCESS SFMs
    'id'    > '\id' store(50) '00' endstore
              store(60) '00' endstore
              use(25)
    'mt'    > next         C An UC ltr should follow these.
    'st'    > next
    'pi'    > next
    'qm'    > next
    'p'     > next
    'r'     > next
    's'     > set(1) use(10)  C Next letter should be UC
    'q2'    > next
    'q'     > next
    'm'     > use(1)       C Next letter need not be UC
    'c '    > store(50) use(50)   C Store these num's
    'c'     > store(50) use(50)   C for the messages
    'v '    > 'v' back(1)
    'v'     > store(60) use(60)
    'e'     > use(1)

group(25)                  C OUTPUT ID LINE
    nl '\'  > next
    '\'     > nl use(20)

group(50)                  C PROCESS & STORE CHAP NO
    any(4)  > dup
    ' '     > endstore append(50) ':'
              store(60) '00' endstore use(1)
    nl      > endstore append(50) ':'
              store(60) '00' endstore use(1)

group(60)                  C PROCESS & STORE VERSE NO
    any(4)  > dup
    ' '     > next
    nl      > endstore if(1) use(10) endif
              ifn(1) use(1) endif
```

2. Before each of the following replacement actions, assume that groups 2 and 10 (in that order) are active. What groups will be active following each replacement action, and in what order?

a. `use(1,10)` _____

b. `excl(2) incl(1)`_____

c. `incl(2)` _____

d. `incl(1,10) excl(2)` _____

e. `use(10,1)` _____

Is there any difference in search order between a. and e. above? Why? _____

Fig. 7. 7CHAR.CCT

```
c    7CHAR.CCT Mod 2       15-JUN-90
c    (Modified for PAD-CCP, May 1991, by K. Seitz)
c    Search for ??? for sections to modify.
c    Deletes words less than 7 chars. from output of WDL
c    Assumptions:
c    1 All refs have been deleted from the file being read.
c    2  "  page headings "   "      "   "    "    "     "
c    3  "  reference counts  "      "   "    "    "     "
c    4 An id line has been inserted in the file being read.

begin       > caseless
      store(Diac)   "'_" '"' C ???diacritics
      store(Wd) ''           C stores the word
      store(Chars)  '0'      C counts characters
      store(TotWds) '0'      C counts total words
      store(DelWds) '0'      C counts deleted words (1-6)
      store(RetWds) '0'      C counts retained words (7 +)
      endstore

            c HOUSEKEEPING
group(1)
    '\id '  > dup use(4)      c retain id line
    '\p'    > ifn(1)          c keep first \p
                 nl '\p '
              endif
              store(Chars)    c set char counter to 0
              '0' endstore
              set(1) store(Wd)
              use(2)
    nl      > next            c create a \p at 1st word
    ' '     > ifn(1)          c create a \p at 1st word
                 nl '\p ' endif
              store(Chars)    c set char counter to 0
              '0' endstore
              set(1) store(Wd)
              use(2)
    endfile > do(Rep)         c EOF is read.

            c ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
            c TEST THE LENGTH OF THIS WORD
group(2)
    any(Diac) > dup           c don't count diacritics
    nl      > next                  c word is over;
    ' '     > endstore incr(TotWds) c word is over;
              ifgt(Chars) '7'       c ??? if over 7,
                 out(Wd) incr(RetWds) c output it,
                 nl
               else
                 incr(DelWds)       c else forget it!
              endif nl back(1) use(1)
    ''      > fwd(1) incr(Chars)    c count this letter
    endfile > endstore do(Rep)
```

(continued on next page)

Fig. 7. 7CHAR.CCT continued

```
            c ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
            c OUTPUTS THE \ID LINE
group(4)
    nl      > dup back(1) use(1)

            c ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
            c OUTPUT THE FINAL REPORT
            c   TO THE FILE AND THE SCREEN.
define(Rep) >
    nl out(TotWds) " total words"
    nl out(DelWds) " deleted words (less than 7 chars)"
    nl out(RetWds) " retained words (7 + chars)"
    endfile
    write  nl
    wrstore(TotWds) write " total words" nl
    wrstore(DelWds)
    write " deleted words (less than 7 chars)" nl
    wrstore(RetWds) write " retained words (7 + chars)" nl
```

3. `List FIXEM.CCT` and look through it. Notice that the group numbers are non-consecutive but are in ascending order. Notice the use of comments to document the purpose of each group as well as other uses.

4. Figure 7. contains 7CHAR.CCT. See if you can draw (in the space below) a block diagram of the flow from group to group, such as that in Figure 5. Again, don't be concerned about the commands we haven't covered. Only look at the `group` and `use` commands.

READING ASSIGNMENT

CC User's Guide: 16 excl, **17** group, **19** incl, **21** name, **25** use, **38-40** groups

Fig. 8. REFIND.CCT

```
C    REFIND.CCT           Mod 1       8-SEP-88
C    (Modified for PAD-CCP, May 1991, by K. Seitz)
C    Extracts all cross references.

begin         > use(1)

group(1)   C ^^^   FINDS SFMS   ^^^^^^^^^^^^^^^^^^^^^^^^^
    '\'       > use(2)        C SFM found!
    endfile   > endfile
    ''        > omit

group(2)   C ^^^   FIND ID LINES and CROSS REFERENCES   ^^^
    'id '     > nl '\id ' use(3)
    'r '      > nl '\r '    use(3)
    endfile   > endfile
    ''        > omit use(1)

group(3)   C ^^^  DUP ID LINES and CROSS REFERENCES   ^^^^^
    nl '\'    > next
    '\'       > nl '\' back(2) use(1)
```

EXERCISES

1. On a separate sheet of paper to be handed in, draw a block diagram of REFIND.CCT shown in Figure 8.

2. Look at WDLSTRIP.CCT (in Fig. 3. and on student disk).
   a. First on paper, then on the computer, modify this file to change every 'a' to an 'x' in the words. Words are marked by \w. These are identified by matching `'w '` in `group(2)`. Put the orthography change ('a' to 'x') in `group(10)`.
   b. Run CC using your table and input file STUDENT.WDL
   c. After successfully completing CC with the desired results, print out your table, put your name on it and hand it in.

3. You have an input file (SOMT.SFM) which must be changed before it can be run through our preprocessing. It contains the following SFMs: \b, \c, \id, \p, \s, \s2, and \v.
   a. Write a table to accomplish the following:
      – Change '\b' to '\mt', and retain all other SFMs
      – In elements marked with \p, \s2, and \v, change '"' to '/', change '-u' to '_u', change '=' to '-', and change '::' to ':'

   (**Hint:** You should be able to do this using two groups. The only replacement actions needed are character strings and group

related commands.)

b. Run CC using your table and input file SOMT.SFM
c. After successfully completing CC with the desired results, print out your table, put your name on it and hand it in.

Mod 5 Using Storage Areas

COMMANDS/TOPICS COVERED

store	**any**
endstore	**command line running of CC**

OBJECTIVES

At the end of this module, the student will be able to:

- modify and/or write a change table, correctly using `any`, `endstore`, and `store` commands;
- run CC, entering all file information needed on one line at the DOS prompt.

INSTRUCTION

1. **Storage Areas—** In Mod 1, you were told that CC produces one output file. This is true. However, you may also create some *temporary* output areas called storage areas in computer memory. This is done with a `store` command which is immediately followed by an identifying name or number. The `store` command is used only on the replacement side:

   EXAMPLE

   ```
   "search argument"   >     store(name)
   ```

   The `store` command redirects all output from this point on until it is stopped with an `endstore` command. After a `store` command is invoked, *all* output—whether from the input file or from a character string in the replacement side of the change table—will go to the storage area indicated and *not* to the output file. It's like switching a train from one track to another.

   Let's look at some examples:

   EXAMPLE 1

   ```
   begin   > store(vowels) 'aeiou' endstore
   . . .
   ```

   EXAMPLE 2

   ```
   group(1)
      "\s" > store(sect) use(2)
   group(2)
      "\"  > endstore "\" use(1)
   . . .
   ```

EXAMPLE 3

```
group(1)
   "\" > endstore use(2)
group(2)
   "c" > store(chpt) use(1)
   "s" > store(sect) use(1)
   "r" > store(xref) use(1)
```

In the first example, a storage area called ‘vowels’ was opened as part of the replacement side of the `begin` command. The character string ‘aeiou’ following the `store` command would be placed into the output. But where is output being directed now? The `store` command has directed output to the storage area called ‘vowels’ instead of the output file. So the string ‘aeiou’ will be placed in the storage area ‘vowels’.

The command `endstore` stops any further output from going to the storage area ‘vowels’ and will send it instead to the output file. This leaves storage area ‘vowels’ containing ‘aeiou’. We’ll discuss how this can be used in a moment.

In the second example, when a `"\s"` is encountered in the input file, a storage area called ‘sect’ is opened and then `group(2)` is made the active group. All output is now being directed to storage area ‘sect’. This means that all input text up to the next backslash will go into storage area ‘sect’ rather than to the output file.

When the next backslash is encountered in the input, then the `endstore` will stop further output from going into the storage area but will send it to the output file.

The third example is like the second except that one of *three* storage areas may be opened depending on the input. Input text following a ‘\c’ will go into storage area ‘chpt’ until the next backslash. Input text following a ‘\s’ will go into storage area ‘sect’ up to the next ‘\’; and input text following a ‘\r’ will go into storage area ‘xref’.

Here are some additional facts about storage areas:

– output can only go to *one* place—either the output file or one storage area;

– a `store` command terminates any previous `store` command which has not been terminated with an `endstore`, as shown in the following example:

```
begin   > store(vowels) 'aeiou'
          store(consonants) 'bcdfghjklmnpgrstvwxyz'
          endstore
```

(Storing into 'vowels' is terminated by `store(consonants)`.)

– a `store` command *clears* the named storage area of any previous contents. Consider the following:

```
store(book,chpt,verse) endstore
```

The effect of the above line would be to clear out the storage areas named 'book', 'chpt', and 'verse'. (These commands, with the three storage names in sequence, are equivalent to saying `store(book) store(chpt) store(verse) endstore` .)

– the storage area name can be any length and may contain alphabetic characters, numbers, or symbols *other than spaces, commas, or right parentheses*;

– storage area names are case sensitive (`store(\f)` and `store(\F)` would refer to two different storage areas);

– a storage area can hold any amount of data, dependent only on the amount of computer memory available;

– the CC program has a limit of 127 storage areas.

2. `any(name)`— There are a number of uses for storage areas. One of them is in conjunction with the command `any` followed by the name of a storage area. The `any` command is used on the search side of a change entry. Each of the characters in the named storage area will be checked for a match in the input. Interpret it as meaning: if the next *single* input character matches *any* single character in the named storage area, you have a match!

EXAMPLE — CV.CCT

```
begin   > store(vowels)   "aeiouAEIOU"
          store(consonants) "bcdfghjklmnpqrstvwxyz"
                            "BCDFGHJKLMNPQRSTVWXYZ"
          endstore
any(vowels)       >  "v"
any(consonants)   >  "c"
```

This table will change the input text into v's and c's showing vowel and consonant patterns. Input text will be matched *one character at a time.* Each alphabetic character will be changed to a 'v' or 'c' and

each non-alphanumeric character such as spaces or punctuation will be passed to the output file unchanged.

The any command can also be used as a part of a longer search argument:

EXAMPLE

```
begin  > store(space) " " nl endstore
"Jesus" any(space) "Christ"    > "Jesucristo"
```

In the above example, input text will now match whether the words ‘Jesus’ and ‘Christ’ are separated by a space or a new line.

The storage area name can also be repeated to indicate two or more successive characters are required to match:

EXAMPLE

```
begin             >  store(space) " " nl endstore
any(space,space)  >  " "
```

Any combination of a new line or space followed by a new line or space would match the above search argument and be replaced by a single space.

3. **Command line running of CC—** Up to now you have run CC by keying CC and pressing <ENTER> and then answering the prompt for the file names. There is another way to run CC. All of the information can be provided on one line at the DOS prompt. Following is the syntax to use.

   C:\> CC -t change table name -o output name input name

   There are two ways this can be beneficial. First, only the most simple CC tables yield the desired results the first time. During the *debugging* of a change table (and, therefore, multiple executions of CC), it is convenient to be able to bring back the last command entered (with all the information needed) by pressing <F3> at the DOS prompt (or the <UP> arrow if you are using NDOSEDIT). This saves a lot of rekeying.

   Secondly, this one-line format for running CC can be used from within a batch file for more automated processing.

   Instead of a single input file, CC will accept a file containing a *list of files* as input to CC by preceding the file name with -i. All of the output will go into *one* output file:

   C:\> cc -t change table name -o output name -i input list

VOCABULARY and CONCEPTS

storage area—a temporary holding place for data in computer memory during the running of CC. When the program ends, information in storage areas is no longer available.

PRACTICE ACTIVITIES and QUESTIONS

1. Below are three `begin` entries with replacement actions. Which (if any) will accomplish the same thing? _____

a.
```
begin > store(1) nl
        store(2) nl
        store(3) nl
        endstore
```

b.
```
begin > store(1,2,3) nl nl nl
        endstore
```

c.
```
begin > store(1,2,3) nl
        endstore
```

2. Included on your student disk is CONVOW.CCT. It is a more sophisticated version of the CV table used in the example in this module. Review CONVOW.CCT—read the comments, think of how it would be modified for different job data, look at the grouped entries. You won't know all the commands, but you should understand some of them.

3. Run CC, providing all the file information on one line at the DOS prompt. Use CONVOW.CCT just as it is and any text file for input.

READING ASSIGNMENT

CC User's Guide: 11 any, **16** endstore, **21** name, **24** store, **28-29** Running CC from Command Line, **30** store & endstore, **31-36** An Example of Storage

EXERCISE

1. Modify CV.CCT (the example used in this module—there's a copy on your student disk) so that it will also change any number (0–9) to an 'n'. Using any input file, run CC with a one-line entry at the DOS prompt. (If you must alter your table and rerun CC, remember to use the <UP> arrow or <F3> to recall the line.) When finished, print out your table and turn it in.

2. EMCHEK.CCT (Fig. 9.—and on your disk) has been written using numbers for names of storage areas. First, run CC with this table as is, using EMLK.SFM on your disk as input. Then modify the

table to use descriptive words for storage area names. Rerun CC giving your output a different name. Compare your outputs. They should compare equal. When finished, print out your modified table and turn it in. (Note: arguments of the `out` command, as will be learned later, are storage area names.)

3. Write a table to change all occurrences of ‘Book of Acts’, whether the words are separated by a space or a new line, to ‘Acts of the Apostles’ with spaces between each word. Run CC using BOOKACTS.TXT as input, print your table, and turn it in.

Fig. 9. EMCHEK.CCT

```
C EMCHEK.CCT Mod 1  13-MAR-84
C Modified for PAD-CCP Mod 5, May 1991, by K. Seitz
C finds sequences of \m followed by no text

begin       > store(1) '00' endstore
              store(2) '00' endstore
              store(3) '1234567890' endstore
              store(4) ' ' nl endstore
              clear(1) use(1)

group(1)
  '\id'     > nl '\id' use(2)
  '\c '     > next
  '\c'      > store(2) '00' endstore  C clear verse number
              store(1) use(4)
  '\v '     > next
  '\v'      > store(2) use(5)
  '\m '     > next
  '\m' nl   > use(3)
  endfile   > ifn(1) 'no \m errors' nl endif endfile
  ''        > omit

group(2)      C Complete id line
  '\'       > nl '\' back(1) use(1)

group(3)      C Check for \pqrs
  '\v'      > dup back(2) use(1)
  '\'       > set(1) '\m error in ' out(1) out(2) nl
              '\' back(1) use(1)
  any(4)    >
  ''        > omit use(1)

group(4)      C Complete ch no
  any(3)    > dup
  any(4)    > ':' endstore use(1)

group(5)      C Complete vs no
  any(3)    > dup
  any(4)    > next
  ''        > endstore use(1)
```

Mod 6 Matches Conditioned by Environment

COMMANDS/TOPICS COVERED

prec	**wd**
fol	**display/debugging option**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of matched *string* versus matched *environment* by correctly providing the output resulting from given table entries and input text;
- modify and/or write change tables correctly using `any`, `prec`, `fol`, and `wd` commands;
- run CC in display/debugging mode and provide answers to questions concerning the displayed information.

INSTRUCTION

1. **Precede, Follow, and Word Commands—** These commands are similar to the `any` command in that they also reference a storage area which must have been previously loaded with characters. Also like the `any` command, these commands are used on the 'search' side, and each occurrence of the commands represent any *one* of the characters in the storage area. But unlike the `any` command, these commands do not actually constitute part of the match but are considered 'environmental' commands. Let's look at each:

   a. **Precede** or `prec(name)`—This command is used along side the character string that is being searched for. For the search string to be considered a match, not only must the search string match the input text starting at the search pointer, but the single character in the data stream *preceding* the matched characters (i.e., the first character left of the search pointer) must also be contained in the named storage area.

   EXAMPLE

```
begin   >  caseless
           store(condition) "aeiou" endstore

"ft" prec(condition)  > "th"
```

   Any 'ft' in the input text which is preceded by a vowel will be changed to a 'th'. Notice that, even though the condition we are

specifying must occur *before* the character string, the command `prec(name)` can be placed either before the character string, or between the character string and the wedge.

b. **Follow** or `fol(name)`—This command works the same way as `prec(name)` except that the input string matching the search string must be *followed* by a character contained in the named storage area. The `fol` command must be placed between the character string and the wedge.

EXAMPLE

```
     begin      >  caseless
                   store(endings) " " nl ".,?!';" '"'
                   endstore
     "ing" fol(endings)      > "in'"
```

The storage area 'endings' is loaded with the characters which would indicate the end of a word. Any word ending in 'ing' will be changed to end in "in' ".

c. **Word** or `wd(name)`—As you have probably guessed, the `wd` command combines `prec` and `fol`. To be considered a match, the input text must match the search character string, and the character preceding *and* the character following the string must be among those contained in the named storage area. The `wd` command must be placed between the character string and the wedge.

EXAMPLE

```
  begin          >  caseless
                    store(boundaries)
                    " " nl ".?!,;''" '"'
                    endstore
  "the" wd(boundaries) >     "a"
```

Every occurrence of the word 'the' would be changed to the article 'a'; the characters 'the' when a part of a larger word (i.e., them, breathe, father) would not be changed.

2. **Differences between** `any` **and** `prec`, `fol`, `wd`— There are a number of differences between `any` and these other commands:

`Any(name)` can be used by itself as the search argument; whereas `prec(name)`, `fol(name)` or `wd(name)` must always be used in conjunction with a search argument.

EXAMPLE

```
any(space)         >  nl
"the" prec(space)  >  nl "the"
```

`Any(name)` is placed within the search argument at the location representing the actual character of input text it is to match; `prec(name)`, `fol(name)` and `wd(name)` must always be placed between the matched sequence and the wedge (`prec` can also be placed before the matched sequence instead).

EXAMPLE

```
"big" wd(space)    >  "large"
"big" any(space) "bad"    >     "big bad"
```

The input character matching the `any(name)` command is a part of the matched string and will be affected by the replacement argument. But input characters represented by the `prec(name)`, `fol(name)` or `wd(name)` commands are *NOT* considered to be a part of the matched string. These preceding and/or following characters only set the *environment surrounding* a match—they are *not* included in the match itself. Therefore, these characters will not be affected by the replacement argument.

This difference is important enough to warrant a closer look.

EXAMPLE 1

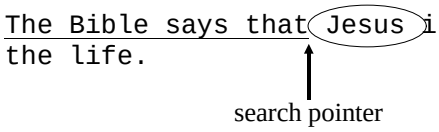
```
  begin               >   store(space) " " nl
                          endstore
  any(space) "Jesus" any(space) > "*Jesus*"
```

EXAMPLE 2

```
  begin               >   store(space) " " nl
                          endstore
  "Jesus" wd(space)   >   "*Jesus*"
```

Input text: `The Bible says that Jesus is the way, the truth, and the life.`

To satisfy the search argument containing `any` in example 1, the search pointer must be positioned as follows; the matched string is circled:



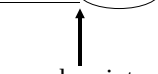
The spaces before and after ‘Jesus’ are *part of* the match. The output file will read:

```
The Bible says that*Jesus*is the way, the truth, and
the life.
```

To satisfy the search argument containing `wd` in example 2, the

search pointer must be positioned as follows; the matched string is circled:

```
The Bible says that Jesus is the way, the truth, and
the life.
```



The spaces before and after 'Jesus' are *not* part of the match. The output file will read:

```
The Bible says that *Jesus* is the way, the truth, and
the life.
```

3. **Display or debugging option** (/d)— As you can see, a Consistent Changes table can easily become rather complex and tricky. Sometimes it's difficult to tell just what entries are being matched, what changes are being made, or what route or path is being followed through the table. Maybe all you know for sure is that you aren't getting the desired results!

   For this reason, the program contains a feature that will let you see on the screen some of the actions which are taking place as the program is running. This is the *display* or *debugging* option. It works best when ANSI.SYS is installed (consult a DOS knowledgeable person about this). The display option is activated by adding /d after the name of the *change table* before you press <ENTER>, or by using the following command at the DOS prompt:

   C:\> CC -t change table name/d -o output name input name

   The program will list on the screen the names of all storage areas, switches*, groups, and defines* in the table (and a number which CC has assigned to each of them). Each name is listed on a separate line. If your table has many, you may need to press <CTRL/S> or <PAUSE>to stop them from scrolling off the screen if you want to see them.

   The display will stop at the first match that is made. For each match, the display will show:

   – the name and contents of any storage area being stored into,

   – one line showing 35 characters of the output and 35 characters of the input—with the matched string in reverse video,

* These features are covered later in the course.

– another line showing the same text after the replacement side has been performed,

– a listing of all the groups that are active and all the switches that are set,

– the name and contents of the currently open storage area (if any) after the match has been completed.

The display will show this same information for each match that is made. Initially, the display will stop at each match until you press any key to continue. But at any stop you may press <ESC> (escape) and it will continue to process matches and scroll the display information on the screen. Pressing any key will again put the action into stopping at each match. Pressing <CTRL/C> will terminate the CC program if desired.

For a visual feature such as this, you will understand it best by using it. Some of the practice activities have been designed to familiarize you with the operation of the display option.

VOCABULARY and CONCEPTS

<CTRL/S>—pressing the <CTRL> key and the ‘s’ key at the same time will stop the information on the screen from scrolling off. Pressing any key will cause the scrolling to resume.

PRACTICE ACTIVITIES and QUESTIONS

1. Using the following change table for each input: 1) draw the location of the search pointer at the time of a match, 2) circle the matched string, and 3) write the final output text.

CHANGE TABLE

```
begin    >  store(space) " " nl
            store(bound) " " nl "’’?!.,;:" ’"’
            endstore

"do" any(space) "not"            >  "don’t"
"already" wd(bound)              >  "all ready"
"semi" prec(bound)               >  "bi"
any(bound) "couldn’t" any(bound) >  "could not"
"wouldn’t" wd(bound)             >  "would not"
```

	INPUT TEXT	OUTPUT TEXT
a.	`I do not care.`	
	search pointer	
b.	`I am already.`	
	search pointer	
c.	`The semiannual news...`	
	search pointer	
d.	`He couldn't go.`	
	search pointer	
e.	`He wouldn't go.`	
	search pointer	

2. Write in the appropriate entries to accomplish the following (including any necessary `begin` entry):
   – change every '(' followed by a number to '[',
   – change every ')' preceded by a number to a ']'.

3. Write in the appropriate entries to accomplish the following (including any necessary `begin` entry):
   – for every ')' followed by a letter or number, output ')***' in place of the ')',
   – for every '(' preceded by a letter or number, output '***(' in place of the '(',
   – for every '(' followed by any type of space, `nl`, or closing punctuation, output '(***' in place of the '('.

4. `List RESPEL.CCT` and look at the uses of `any` and `fol` in groups(40) through (49).

5. Run CC in display mode using RESPEL.CCT as the change table and RESPEL.TXT as the input. Be ready to press <CTRL/S> or <PAUSE> *immediately* after entering the change table name followed by /d. (If you weren't fast enough, use <CTRL/C> to abort the program and try again.) Answer the following questions:
   a. How many stores are there? _____
   b. What is the maximum number of changes allowed? _____
   c. List the first five group sets that become active.
      _____
   d. When an entry containing `fol` is matched, is the character satisfying `fol` (as in `El'ia fol(endword)`) highlighted?___
   e. Is the character satisfying `any` (as in `Eja any(sp) cumu`) highlighted? _____

READING ASSIGNMENT

CC User's Guide: 17 fol, **23** prec, **25** wd, **27** Debug

EXERCISES

1. Combine the table entries you wrote for Practice Activities 2 and 3 into one table. Run CC in display mode using this table and input file PAREN.TXT. Check your results. When finished, print your table and turn it in.

2. Modify RESPEL.CCT to make the following additions:
   (**HINT:** read carefully the comments preceding `group(40)` & `(70)`.)
   a. change 'yyy' to 'zzz' when it occurs at the end of a word;
   b. change 'xx' to 'hx' when it occurs at the beginning of a word;
   c. change the word 'text' to 'test';
   d. There is already an entry for `"p-uga" any(sp) "ni"`. Add a similar entry that will match on the same string except that it will also find any 'endword' punctuation between 'p-uga' and the space. Don't alter the replacement.

   Run CC using RESPEL.TXT as input; check your results; print your table and hand it in.

Mod 7 Getting it Out of Storage

COMMANDS/TOPICS COVERED

out **outs** **append**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of `append`, `out`, and `outs` by identifying the correct output from specific table entries and input text;
- modify and/or write tables correctly using `append`, `out`, and `outs`.

INSTRUCTION

1. **Outputting storage area contents—** So far we have discussed how to direct output text to a storage area, and some ways to compare input text to the contents of a storage area. Now we'll show you how to direct the contents of a storage area to the output file or to another storage area.

   a. `out(name)`—The `out` command will cause the contents of the named storage area to be placed into the output file. It does *not* clear the storage area, but leaves the contents unchanged. You might think of it as putting a copy of the storage area contents into the output file. The `out` command *stops* any current storing and switches the output flow to the output file. Since it is an action to be taken, it is used on the replacement side.

   Let's consider what is needed to handle text where dropped chapter numbers are embedded within the first two lines of the paragraph. Here the chapter must follow any section head which might precede that paragraph. This would require changing the sequence of chapter numbers and section heads, as illustrated in the following example:

EXAMPLE (Place chapter numbers AFTER section heads for dropped chapter numbers)

```
group(1)
  "\c " > store(chpt) "\c " use(2) c store chpt;chk next

group(2)
  "\s " > endstore "\s " use(3)   c output sect. first.
  "\"  > out(chpt) "\" use(1)     c chpt not followed by
                                  c sect—output chpt
group(3)
  "\"  > out(chpt) "\" use(1)     c put chpt after sect.
```

In this example, a storage area named ‘chpt’ is opened each time a ‘\c ’ is encountered in the input text. A ‘\c ’ is output to the storage area and `group(2)` becomes active. All text following the SFM (a chapter number and new line) up to the next backslash also goes to the storage area.

If the next text element is a section head ‘\s ’, then `endstore` sends all subsequent output to the output file. A ‘\s ’ is output, and `group(3)` becomes active. There the rest of the section head is output until the end is found at the next ‘\’. Then `out(chpt)` copies the chapter element stored in ‘chpt’ to the output file, and searching continues in `group(1)` for ‘\c ’.

If the text element following the chapter number is not a section head (\s),then the input text will match on the single backslash in `group(2)`. At this point `out(chpt)` ends storage and copies the ‘chpt’ contents (‘\c ’, a chapter number, and new line) to the output file, keeping the original order as in the input text. Searching continues in `group(1)` for ‘\c ’.

b. `outs(name)`—This command is identical to the `out` command except that it does *not* stop any storing which is currently taking place. In other words, it does not switch the flow of the output. One of the important aspects of this is that it allows you to move data from one storage area to another.

EXAMPLE (Set up ending and beginning chapter number storage areas)

```
group(1)
  "\c "  >  store(oldchp) outs(newchp) store(newchp)
  "\"    >  endstore
            "@hdr endchp = " out(oldchp)
            "@hdr newchp = " out(newchp)
```

When a new chapter number is encountered (\c), the character string stored in the current chapter storage area (newchp) is transferred to the old chapter storage area (oldchp). This transfer is a two-step process: first, the `store(oldchp)` command switches the output flow to the ‘oldchp’ storage area; then the `outs(newchp)` command copies the ‘newchp’ storage contents into the output flow, putting it into ‘oldchp’.

If the `store(oldchp)` command had not been given, the `outs(newchp)` command would have sent the ‘newchp’ storage contents to the output file. Note that if `out` had been used instead of `outs`, it would not have made any difference whether the `store(oldchp)` command was there or not because `out` would have terminated the `store` before outputting anything. So the

'newchp' contents would have gone to the output file.

In the example as it is written, what terminates the `store(oldchp)` command? (Answer: `store(newchp)`)

2. **Adding on to storage with** `append(name)`— The `append(name)` command is quite like the `store(name)` command except that it does *not* clear out the storage area. It can be used to *add* more text to the contents of a storage area.

   The following example is an abbreviated change table to be run on Scripture text. Expected SFMs in the input include \id (followed by a three-character book name and possibly other information), \c, \s, \p, and \v. Any other SFM will be considered an error. The output will be all of the input text plus error messages for unexpected SFMs—giving the book, chapter, and verse where it was found. This example shows a use of the `append` command as well as additional uses of `out`:

EXAMPLE (to reference error messages)

```
group(1)
  "\id "  > "\id " store(book) use(2)   c begin book store
  "\c "   > "\c " store(chpt) use(3)    c begin chpt store
  "\s "   > "\s "                       c protect known SFM
  "\p" nl > "\p" nl                     c protect known SFM
  "\v "   > "\v " store(verse) use(4)   c begin verse store
  "\"     > "***unknown SFM found at "  c output error msg
            out(book,chpt,verse)        c  for unknown SFMs
            "***" nl "\"                c  with reference

group(2)          c  find end of \id book name
  " "     > " " endstore                c output book name
            out(book) use(1)

group(3)          c find end of chapter number
  nl      > endstore
            out(chpt) nl                c output chapter #
            append(chpt) ":" endstore   c add colon to chpt
            use(1)                      c  for error msg

group(4)          c find end of verse number
  " "     > endstore out(verse) " "     c output verse #
            use(1)
```

Your practice activities will make use of this example to help you understand it better.

PRACTICE ACTIVITIES and QUESTIONS

1. Using the preceding example, explain what will happen when '\q' is encountered in the following input text:

INPUT

```
\id MAT Example
\c 1
\s The Genealogy of Jesus
\p
\v 1 A record of the genealogy of Jesus Christ, the son
of David, the son of Abraham:
\q
\v 2 Abraham was the father of Isaac, ...
```

Write out below what the output would be for this segment of text.

_____

_____

_____

_____

_____

_____

_____

_____

_____

2. Run CC using table MOD7PRAC.CCT (a disk copy of the above example) with MOD7PRAC.SFM as input, and check the output against your answer for 1. above. (If it is different, try running it in display mode and stepping through it.)

3. '\q' is a legitimate SFM; write in below the entry needed to treat it the same as a '\p'.

_____

READING ASSIGNMENT

CC User's Guide: 11 append, **22** out & outs, **30** append, **31** out & outs

EXERCISES

1. Fig. 10 below shows the part of a table which prepares verse numbers for Ventura. What will be the output for the following texts (write your answers on the paper for Exercise 2):

INPUT	OUTPUT
\v 2<space>	_____
\v 3-5<space>	_____
\v 6a<space>	_____
\v 6b-7<space>	_____
\v 10,11<space>	_____
\v 42<space>	_____

2. We want to know the sequence of the SFMs in an input file. Write a table that will do the following:
   a. output all of the id line (from \id to the next \);
   b. at \e, output 'End of Book ';
   c. at \c, \mt, \m, \p, \q, \r, and \s, accumulate in a storage area the SFM (without the backslash) that was found, followed by a hyphen (e.g., mt-c-s-r-p-...). These should continue to add on until step d. below;
   d. at \v, output the stored sequence, a `nl`, and clear the store;
   e. strip all other text. (This requires a couple commands we haven't covered, so group(1) is provided for you below.)

```
group(1)
'\'     > use(2)    c finds start of SFM
endfile > endfile   c terminate program at end
""      > omit      c strips all else
```

   f. Run CC using this table and HDJN.SFM for input. Print your table and hand it in.

Fig. 10. Verse Number Attributes for Ventura

```
begin > store(NBHy) "_"    c Underline; NoBreakHyphen
        store(VP) "9"      c Verse Point Size
        store(VJ) "250"    c Verse baseline Jump
        store(sp) " " nl   c Space characters
        endstore
...
group(1)
'\v '   > store(VsNum)
          use(3)
group(3)
'-'     > append(VsNum) outs(NBHy)
any(sp) > endstore
          '<BP' out(VP)
          'J' out(VJ) '>'
          out(VsNum) '<PDJ0>' use(1)
```

Mod 8 Moving Text On Through

COMMANDS/TOPICS COVERED

dup	**endfile**	**null**
next	**omit**	

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of the use of nulls by identifying right and wrong table entries;
- modify and/or write tables correctly using `dup`, `next`, nulls, and `omit`;
- write a statement in his own words on how to avoid getting into a loop with a null search.

INSTRUCTION

1. **Duplicating the matched string into the output with** `dup`— Up to now, if we wanted matched text to be sent to output we accomplished this by repeating the matched character string on the replacement side of an entry. But there is an easier way.

   When input text matches a search argument, not only is it removed from the input text, but it is also copied to a "match buffer." Every time you use `dup` within that replacement argument, a copy of the match buffer is sent to the output. As implied, this command is used only on the replacement side.

   EXAMPLE 1

```
group(1)
"\c"    > dup use(2)
```

   Example 2

```
begin                           > store(space) " " nl
                                  endstore
"Jesus" any(space) "Christ"     > store(text) dup
                                  endstore
```

   In the first example, matching on '\c' is used to trigger a change in active groups. The matched string (\c) is sent unchanged to the output file with `dup`.

   In the second example, because of the use of `any`, the exact matching text is unknown. There could be a space or a new line between the words. In order to send it to output unchanged, `dup` is essential.

Notice that in the replacement argument in this example a storage area is first opened, and then the matched string is copied into it from the match buffer using the `dup` command.

Remember in using the commands `prec`, `fol`, or `wd` that the characters required by these commands are not a part of the matched string and therefore *cannot* be placed into the output using the `dup` command.

EXAMPLE

```
begin                 >   store(wdbound) " " nl ".,;''!?"
                          endstore
"Jesus" wd(wdbound)   >   dup
```

Only the word 'Jesus' will be sent to output, and *not* the character on either side of it.

2. **A simple shortcut using** `next`— Sometimes you may have a number of consecutive search entries which all have the *same* replacement argument. Rather than entering the same replacement side for each line, the `next` command may be used for the replacement side for all but the last one. Remember, they must be consecutive and the replacement argument identical.

EXAMPLE

```
begin             >  store(sp) " " nl endstore
any(sp,sp,sp)     >  next
any(sp,sp)        >  next
any(sp)           >  nl  "The next word is:"
```

Whether 1, 2, or 3 spaces are encountered, the replacement will be a 'new line' and the message 'The next word is:'.

3. **Recognizing the end of file with** `endfile`— It is often useful to detect the end of the input text. This is done by using the `endfile` command on the search side of an entry to match on the 'end of file'. Two principle reasons for using the `endfile` command are:

   a. performing finishing up tasks at the end of input files, such as report summaries or outputting the remainder of a record, and

   b. preventing infinite loops where null matches are used, as described later in this module.

   The replacement argument may contain actions to take before the program stops, including outputting messages or doing other finishing tasks. The replacement side must either contain an `endfile` to output an 'end of file' or a `use` command that will transfer control to

another group where an `endfile` will be output. The program will not end after matching on `endfile` until an `endfile` is output.

The simplest usage would be:

```
endfile     >    (actions if needed)  endfile
```

On rare occasions it may not be necessary to process all of an input file. In such cases, `endfile` can be used as a replacement argument to end the program *before* the end of file is reached on the input file.

4. **Bypassing or dropping text with** `omit`**—** There are times when it is necessary to eliminate a certain number of input characters from the data stream without being processed. The `omit` command removes characters from the input immediately to the right of the search pointer and discards them. They will not be available for matching at a later time.

   The `omit` command is followed by a number in parentheses, specifying the number of characters to be omitted. When `omit` is used without such a parenthesized number, the number *one* is the default. This command is used *only* in the replacement side.

   EXAMPLE

   ```
   "***"     >        omit(50)
   ```

   The above line would cause the next 50 characters following *** to be omitted from the processing and from the output.

5. **The use of nulls—** A ‘null’ in CC is written as two delimiters with nothing in between: `''` or `""`. It has a use on both the search side and on the replacement side. If certain cautions are not heeded, it can cause an endless loop when used on the search side.

   a. Replacement side null—A null on the replacement side simply means that a ‘nothing’ will be output. It is usually shown on the replacement side only for human readability.

      EXAMPLE

      ```
      "text string" >   "" use(10)
      "text string" >   use(10)
      ```

      Both of the above lines have the same result—in each case, nothing is sent to output.

   b. Search side null—Using a null as a search argument means “match on ‘nothing’”. A null (or ‘nothing’) has a length of zero characters, therefore, CC will attempt to match it last when all en-

tries are sorted by length. A null will *always* match the zero-length ‘nothing’ between the search pointer and the first character to the right of it.

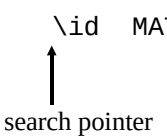
EXAMPLE **(drop everything but the section heads)**

```
group(1)
  "\s"        >   dup use(2)
  ""          >   omit(1)
group(2)
  "\"         >   use(1)
```

In `group(1)` of the above example, unless the two characters to the right of the search pointer are ‘\s’, the null search entry will match on a ‘nothing’ to the right of the search pointer.

EXAMPLE

Input text:



The data stream would first be compared with ‘\s’. Since that would not result in a match, the data stream would then be compared with a ‘nothing’. The ‘nothing’ between the pointer and the ‘\’ *does* match, the ‘nothing’ is removed from the data stream causing the pointer to (still) point before the ‘\id...’, and the replacement instructions are performed. The first instruction in the replacement is `omit(1)`, which causes one character to the right of the search pointer (the ‘\’) to be removed from the data stream. With the search pointer now pointing before the ‘id...’ the searching for a match continues.

c. Nulls and `endfile`—After all the input text has been read, there will still be one more character, the *end of file,* in the input. Here it is necessary to prevent a null search entry from matching the null located between the search pointer and the end of file character, since a null search entry will match *anywhere*. This would be an appropriate time to use the `endfile` entry described above to match the end of file and send an `endfile` character to the output, ending the program.

d. Nulls in multiple search groups—When searching for a null while multiple groups are active (e.g. `use(1,10,20)`), be sure *only* the last active group has the null entry. If it is located in any other group, subsequent groups will *never* be searched since the null will have already been matched.

e. **CAUTION ! ! ! The endless loop**—As just seen, when a null search argument matches on a 'nothing', no forward progress is made by the search pointer. Therefore, to prevent the null search from matching at the same spot again *(and again, and again...)*, you must either: 1) include a replacement side command such as `omit`, or `fwd` (see next module) that forces the pointer to move, *and* an `endfile > endfile` entry to terminate the program should the input data run out, or 2) include a replacement side `use` command to switch groups to one where a valid match can occur.

Otherwise, the program may be stuck in an endless loop with no escape but <CTRL/ALT/DEL>!

VOCABULARY and CONCEPTS

<CTRL/ALT/DEL>—Pressing <CTRL>, <ALT>, and <DEL> keys at the same time will cause the computer to reinitialize. This is called a warm boot.

loop—When a program's logic leads it into a circular path, executing the same set of instructions over and over with no change or advancement, it is in a loop. It will continue cycling through those same instructions without end. The program must be terminated with <CNTL/ALT/DEL>. Then find and change the erroneous logic in the change table and rerun CC.

PRACTICE ACTIVITIES and QUESTIONS

1. Write an entry that will force a match and remove the next character from the input if no other entry matches the data.

_____

2. Write an entry that, when used along with a null match entry, will match at the end of the input file and terminate the CC program, thus preventing an endless loop.

_____

3. Write the replacement side which will cause the match to be written to the output.

   `any(sp) >` _____

4. Will any of the following entries cause a problem? Why?
   a. `"" > omit`
   b. `"" > use(2)`
   c. `"" > ""`

_____

_____

_____

5. `List TAGTOSFM.CCT` and look at the uses of `dup`, `next`, nulls, and `omit`.

READING ASSIGNMENT

CC User's Guide: 15 dup, **16** endfile**, 21** next, **21** null match or replacement, **22** omit, **34** endfile (bottom of page)

EXERCISES

1. Write a table which will do the following:
   – write the \id line to output;
   – strip all other SFMs (but not the text following them);
   – strip all Ventura tags (i.e., strip all strings that begin with '@' and end with '<space>=<space>');
   – strip all Ventura text attributes (i.e., strip all strings that begin and end with angle brackets: '< . . . >').

   Use two groups. One of the groups will contain all of the 'end of string' elements (i.e., ' = ', '>', ' ', and 'new line').

   Use nulls, `dup`, `next`, and `omit` to the fullest extent.

   Run CC using FRUME.TXT as input, and check your output.

   When finished print your table, and hand it in.

2. Modify your change table written for Exercise 1 to replace the '\p ' SFM and the '@par = ', '@par–fol–sect = ', and '@par–fol–chp = ' Ventura tags with '**para** ' (while still stripping all other SFMs and tags). Again run CC using FRUME.TXT as input and check your output. Print your table and hand it in after completing Exercise 3.

3. At the bottom of your Exercise 2 printout, write out a statement from memory of how to avoid getting into an endless loop with a null search.

Mod 9 Going To And Fro

COMMANDS/TOPICS COVERED

fwd	**what can be searched for**
back	**what can be sent to output**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of `back` and `fwd` by answering questions concerning specific input and table entries;
- modify and/or write tables correctly using `back` and `fwd`;
- list from memory at least 3 of the 4 types of items which can be searched for and 4 of the 5 types of items which can be sent to output.

INSTRUCTION

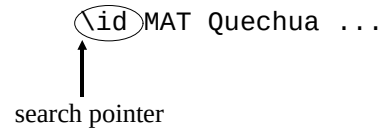
1. **Sending input directly to output with** `fwd`— The `fwd` command is quite similar to `omit` except it *forwards* the number of input characters specified in its parenthesized argument directly to output rather than omitting them. When no number is specified following `fwd`, the number one is the default.

   To illustrate, look at the command `fwd(3)`. This command causes the first three characters to be removed from the input (the right side of the search pointer), and placed unchanged into the output (the left side of the search pointer).

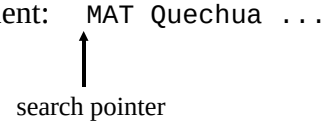
   EXAMPLE

   ```
   "\id " > fwd(3)   c forward book name to output
   ```

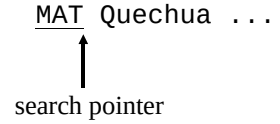
   Data stream at time of match:

   After match, before replacement:

   After replacement:

The most useful purpose of the `fwd` command is when it is used in the replacement side of a null match to move one character of the input directly to the output. The search pointer essentially moves one character, thus preventing an endless loop.

2. **Retrieving output with `back`—** In the same manner as `fwd` sends input directly to output, `back` sends the specified number of characters of the *output* back into the *input* for reprocessing. The characters will be removed from the output just to the left of the search pointer and placed unchanged in the input just to the right of the search pointer. Again if no number of characters is specified with `back`, the default is one.

   Let's look again at an example similar to the one used in Mod 7:

EXAMPLE

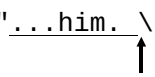
```
group(1)
 "\id " >  dup store(book) fwd(3)  c begin book store
           " " endstore out(book)
 "\c "  >  dup store(chpt) use(2)  c begin chpt store
 "\s "  >  dup                     c protect known SFMs
 "\p "  >  dup
 "\v "  >  dup store(verse) use(3) c begin verse store
 "\"    >  "Unidentified SFM found at " c send error msg
           out(book,chpt,verse)    c  for unknown SFM
                                   c  with ref
group(2)
 "\"    >  endstore out(chpt)      c end chpt store
           append(chpt) ":"        c add colon after
           endstore                c  chpt no.
           dup back(1) use(1)      c retrieve backslash
group(3)
" "     >  endstore out(verse)     c keep verse no.
           dup use(1)              c in output
```

The last line of `group(2)` uses the `back` command. Why is it used here? It is because the '\' entry in `group(2)` will match the first backslash in the input text following a chapter number element, signaling the start of a new SFM. Once it is matched, the backslash is *removed* from the input text and is unavailable for further matching. Unfortunately, it is still needed in the input string in order to correctly identify the subsequent SFM in `group(1)`!

To make the '\' available for matching again, it must *first* be placed into the *output* by the `dup` command so that it can then be *retrieved* from the output and returned to the *input* with `back(1)` for reprocessing.

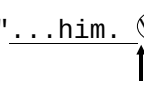
Let's step through the commands in this example table using the

data stream `"...him. \c 2\s Jesus..."`, starting with the search pointer positioned before the chapter SFM while `group(1)` is active:

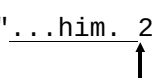


Again the characters to the right of the search pointer in this data stream are input text characters, copied into the data buffer from the original input file. The underlined characters to the left of the pointer are output text characters, which will ultimately be copied to the output file.

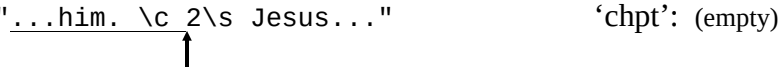
Searching the entries in `group(1)`, the `"\c "` entry will match the input text:



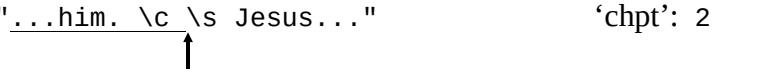
The matched text is removed from the data stream:



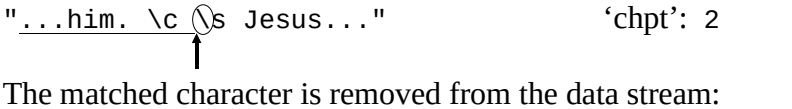
In the replacement for `"\c "`, `dup` will copy the ‘match buffer’ to the output, `store(chpt)` will divert all future output to the ‘chpt’ storage area, and `use(2)` updates the currently active groups list:



With that replacement finished, CC is ready to do a new search. Checking the active group list it finds `group(2)` active. However, it does not find a match in `group(2)`. Therefore exactly one character is removed from the input (*as though* it had matched) and an identical character is placed in the output (which has been redirected to ‘chpt’!):



Before looking for the next match, the active group list must be consulted, and it says to (still) use `group(2)`. The ‘\’ entry matches the input:



The matched character is removed from the data stream:

```
"...him. \c s Jesus..."                'chpt': 2
```

Output is restored to the data stream by `endstore`, and `out(chpt)` copies the contents of 'chpt' to the output:

```
"...him. \c 2s Jesus..."               'chpt': 2
```

Output is once again diverted to the storage area 'chpt' by `append(chpt)` *without erasing* its contents. A ':' is output to it, then output is restored to the data stream by `endstore`:

```
"...him. \c 2s Jesus..."               'chpt': 2:
```

The 'match buffer' containing '\' is copied to output by `dup`:

```
"...him. \c 2\s Jesus..."              'chpt': 2:
```

One character is removed from output and placed in input by `back(1)`, and the active groups list is updated by `use(1)`:

```
"...him. \c 2\s Jesus..."              'chpt': 2:
```

This results in the search pointer now being positioned before the backslash in the data stream, ready to be compared to the search entries in `group(1)`. The output text still contains the same sequence `"\c 2"` that previously existed in the input, while the 'chpt' storage area is now updated with the current chapter number, including a colon, for use at later stages of the text analysis.

This level of detail may seem excessive but a good understanding of the flow of text through the processing will be helpful in debugging complex tables.

3. **Summary of types of items in search and replace sides—** Perhaps it's time now to list specifically *all* the types of things which can be *searched for* and things which can be *sent to* the output file. We've already discussed most of them.

   a. What can be searched for?
      1) any character or string of characters within delimiters, including spaces and tabs,
      2) `nl`, which will match on line endings,
      3) characters in a storage area through the use of commands like

`any`, `wd`, `fol`, etc.

4) ASCII numbers. Any character, which cannot be represented by a 'key top' character on the keyboard (such as non-printing or upper ASCII codes) can be represented by its ASCII code. Any number outside of delimiters will be assumed to be an ASCII code. If a 'd' immediately precedes the number, the number will be treated as a *decimal* ASCII number. With no letter preceding the number, it will be assumed to be octal.

EXAMPLE

```
"   " > d9        c replace 3 spaces with tab

10    > ""        c remove backspace
```

b. What can be sent to the *output* file?
   1) character strings within delimiters
   2) `nl`, i.e., a <CR/LF> character
   3) ASCII numbers (outside of delimiters)
   4) contents of storage areas by using commands `out` and `outs`
   5) All unmatched characters not specifically omitted

VOCABULARY and CONCEPTS

ASCII numbers—(American Standard Code for Information Interchange) A numbering system used by computers. A number is assigned to each character or control function. For the octal, decimal, or hexadecimal representation of these numbers and the assigned characters see the chart at the end of the CC User's Guide.

PRACTICE ACTIVITIES and QUESTIONS

1. After matching on a backslash, we want to make it available again for another match. Will the following entry do this? Why or why not?

```
"\" > back(1) use(2)
```

---

2. Write in the correct entry to send the SFM and three-character book name directly to the output unchanged, after matching on `"\id "`.

---

3. Beginning with the following data buffer and search pointer position, perform the match described in the table entry, then write the resulting data buffer and search pointer position:

DATA BUFFER

```
\s ??Be'taj okme'dik ya'e? \r [Lc. 12.2-9]
```

TABLE ENTRY

```
"[" > "(" back(1) use(40)
```

RESULTING DATA BUFFER

_____

READING ASSIGNMENT

CC User's Guide: 8 ASCII codes, **12** back, **17** fwd, **36-38** back

EXERCISES

1. a. Write a "cleanup" table that will accomplish the following:
   – change multiple spaces into single spaces;
   – change multiple new lines into single new lines;
   – remove spaces from the beginnings of lines and from the ends of lines;
   – remove spaces before backslashes;
   – ensure that a new line precedes every backslash.

_____

_____

_____

_____

_____

_____

_____

   b. Key in your table and run CC using SLOPPY.SFM as input. Check the results.

   c. Print the table and hand it in after completing the following Exercise 2 and 3.

2. From memory list at least 3 of the 4 types of items which can be searched for and 4 of the 5 types of items which can be sent to output. After writing your answers on the following lines, transfer your answers to the bottom or back of the Exercise 1 printout.

a. Types of items which can be searched for:

_____

_____

_____

_____

b. Types of items which can be sent to output:

_____

_____

_____

_____

_____

3. `List CSTCHK.CCT`; make a note of the use of `store(1)` and `store(2)` in the beginning comments; search for `define(2)`. Below your Exercise 2 answer on the printout, write out what the replacement action of `define(2)` is doing. (You are not expected to understand the meaning of `define` or of the storage contents, only describe what is being sent to output.)

Mod 10 Introduction to Switches

COMMANDS/TOPICS COVERED

switches	**if**	**endif**
set	**else**	**visual alignment**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of the concept of switches by identifying and providing some common decisions which fit the concept of switches;
- modifying and/or writing change tables correctly using `set`, `if`, `else`, and `endif`;
- visually align the components of a change table to aid human readability.

INSTRUCTION

1. **The concept of switches—** One of the strengths of Consistent Changes is its ability to make changes based on the *context* of the match. This can be accomplished in several different ways. One way is to set a *switch* when a certain condition is encountered, and then to *test* whether that switch is on or not at the decision point where the change is to be made.

   Perhaps the most familiar switch to compare the concept to would be an electric light switch in a room. The logic might include: when someone enters the room, the light is turned on; later, when someone else comes along, he can decide whether to go into the room to see the first person by whether the light is on or not.

   Now let's put this example in the structure more like we would deal with in CC. We find an input match (person A) so we turn the switch on. Later we have another input match (person B). The replacement action is *contingent* upon whether person A preceded, so we test the switch. If the switch is on, person B will go into the room. But if it is not on, person B will go elsewhere.

   The rest of this module and the next will provide you with the commands and syntax for using switches.

2. **Turning switches on with `set(name)`—** Like most light switches, CC switches have two states: *on* and *off*. A switch is turned 'on' with

the `set` command, immediately followed by the name of the switch in parentheses. (The same rules apply to switch names as apply to groups and stores.) Because a switch will be turned on as a result of an input match, we know that the `set` command will be used only on the replacement side.

EXAMPLE

```
group(1)
"\c"          > set(chpt) use(2)
...
```

Here a match on '\c' will turn *on* a switch named 'chpt' for later testing.

3. **Testing a switch with `if(name)`—** In our original example we might have stated this step as: *if* the light is on, go into the room. In other words, the action of going into the room will be taken only if the light is on.

   Let's pursue our CC example that was presented above:

   EXAMPLE

```
group(1)
"\c"          > set(chpt) use(2)
"\s"          > if(chpt) "@sect fol chpt = "
...
```

   When '\s' is encountered in the input, we want to know if the section head is immediately following a chapter number. If it is, we will output a Ventura tag that has appropriate spacing for a section head following a chapter number.

4. **Stating the alternate action with `else`—** What happens if the switch is *not* on? Sometimes nothing—the action that would be taken if the switch were on fails to occur, and that is all that is required. But other situations call for one consequence if the switch is on and another consequence if the switch is off. 'If the light is on, go into the room; otherwise, go watch TV'. For our CC example:

   EXAMPLE

```
group(1)
"\c"          > set(chpt) use(2)
"\s"          > if(chpt) "@sect fol chpt = "
                    else
                           "@sect hd = "
```

   One Ventura tag (`@sect fol chpt = `) is output if the switch is on, but a different tag (`@sect hd = `) is output if the switch is not on.

5. **Ending the conditional action with `endif`—** Most `if` statements will require an `endif`. This is necessary for the program to know how

many actions to skip if the switch did not have the right setting for those actions to be executed.

EXAMPLE

```
group(1)
"\c"        > set(chpt) use(2)
"\s"        > if(chpt) "@sect fol chpt ="
                       set(\c\s)
                 else
                       "@sect hd = "
                 endif
                 set(sect)
                 use(3)
...
```

Now, if ‘chpt’ switch is on, a tag name is written to output and another switch is set—indicating that a chapter number followed by a section head has been encountered. The `else` establishes a boundary on the actions to be taken if ‘chpt’ switch is on. If ‘chpt’ switch is off, a different tag is written out, and the boundary of the ‘off’ consequences is established by `endif`.

What about the two actions following the `endif`? They are unconditional actions to be taken regardless of ‘chpt’ switch settings. Without the `endif`, these actions would be indistinguishable from the ‘off’ actions.

Unconditional actions (actions to be taken regardless of switch conditions) can be placed either before the `if` command or after the `endif`. Sometimes it may be necessary to perform an unconditional action *before* the `if`; other times an action may be required *after* the `endif`. Study the following example:

EXAMPLE

```
group(1)
"\c"           > set(chpt) use(2)
"\s"           > store(sect)
                 if(chpt)  "@sect fol chpt ="
                           set(\c\s)
                  else
                           "@sect hd = "
                 endif
                 endstore
                 set(sect)
                 use(3)
...
```

Note that the `endstore` command would produce different results had it preceded the `endif` command.

Let's look at one final example where an `else` is not needed:

EXAMPLE

```
        ...
        endfile       > if(paren)
                           '***FINAL PAREN UNMATCHED IN'
                           out(book) nl
                        endif
                        endfile
```

Here the end of the input file has been encountered and a closing parenthesis was never found to match the last opening parenthesis. An error message is generated. The `endif` forms the limit of actions to be taken if the 'paren' switch is on even though there is no `else`. The `endif` is needed so that `endfile` will be output whether the 'paren' switch is on or off.

6. **Testing for multiple conditions—** Frequently it is necessary to test two or more switches to determine if an action needs to be done. Multiple `if` commands together can be used to accomplish this.

EXAMPLE

```
        "\s"          > if(chpt) if(intro) "@sect int = "
                        endif
                        set(sect)
                        use(3)
```

In this example, the Ventura tag `"@sect int = "` will be output only if *both* switches 'chpt' *and* 'intro' are on. Also, the `endif` command ends *all* `if` conditions currently in effect so that `set(sect) use(3)` will be performed unconditionally.

7. **Alignment hints for readability—** There are certain alignment 'conventions' which will greatly enhance readability when followed:

EXAMPLE

```
group(1)
"search string" >  actions                      c comments
                   if(name)                     c comments
                         actions (incl. strings) c comments
                   else
                         actions (incl. strings) c comments
                   endif
                   other actions (incl. strings) c comments
"search string" >  actions                      c comments

group(2)
any(name) "txt" >  actions                      c comments
```

A scheme such as this will greatly assist you or someone else in following the flow of logic through a table.

PRACTICE ACTIVITIES and QUESTIONS

1. Which of the following decisions fit the concept of switches?
   __ a. If the manuscript is complete, we will publish it.
   __ b. If the text contains Standard Format Markers, we'll use CC.
   __ c. Because the printer is broken, we can't complete the job.
   __ d. We will recover the file, if we have the backup disk.
   __ e. On Monday afternoon, we will have a meeting.
2. Some translators will code into their text a second (or third) '\s' to force a line break in a long section head. Write the table entry to replace all but the first '\s' in a section head with a Ventura line break <R>. (Assume that the switch is cleared elsewhere in the table.)

3. Rewrite the following entry for better visual alignment.

```
"\c " > if(\c) endstore "duplicate \c found at "
        out(bk,chpt) use(4) else set(\c) store(\c) dup
        back(3) use(7) endif incl(10)
```

READING ASSIGNMENT

CC User's Guide: 15 else, **16** endif, **18** if, **24** set, **40-44** switches

EXERCISES

1. In Mod 9, Exercise 1, you wrote a table to ensure that each '\' started on a new line. Most likely it resulted in an extra new line before the first '\' in the file. Rewrite your table using a switch to determine if the backslash ought to have a new line before it or not.

_____

_____

_____

_____

_____

_____

_____

_____

_____

   Run CC using SLOPPY.SFM; check your results; print and hand in your table, after completing Exercise 2.

2. On your Exercise 1 printout, write a single entry that, upon matching `'\'`, will do *all* of the following:
   a. put it to output and back up over it;
   b. activate group(2) if switch '\c' is on;
   c. activate group(3) if switch '\s' is on;
   d. activate group(4) if switch '\r' is on;
   e. activate group(5) if switch '\p' is on.

Mod 11 More On Switches

COMMANDS/TOPICS COVERED

clear	**??? search technique**
ifn	**mark & rewind technique**

OBJECTIVES

At the end of this module, the student will be able to:

- modify and/or write tables correctly using `clear` and `ifn`;
- modify a table using ??? search technique;
- modify a table using mark and rewind technique.

INSTRUCTION

1. **Turning the switch off with** `clear(name)`— Returning to the example first used at the beginning of Mod 10 (the light switch), this will only work if person A remembers to turn the light off when he leaves the room. (I seem to recall my mother harping on that!) In CC, a switch is turned off by the `clear(name)` command.

   EXAMPLE

```
"("       >  set(paren) dup
")"       >  if(paren)
                    dup clear(paren)
                 else
                    "UNMATCHED )"
             endif
endfile   >  if(paren)
                    "UNMATCHED ("
             endif
             endfile
```

   In this example, a switch is set when an open parenthesis is found. When a closing parenthesis is encountered, the switch is turned off if it is on. If it is not on, an error message is put to output. If the switch is not turned off after a close parenthesis is processed, then subsequent close parentheses would not generate error messages.

2. **Testing for an off condition with** `ifn(name)`— If consequences were only to be performed if the switch was off, it would be possible to write the entry:

EXAMPLE

```
"string"          >  if(name)        c (do nothing)
                       else
                          "consequence"
                     endif
```

However, we have been provided a more straight forward way of doing this with the `ifn(name)` command. The literal reading of this is: if switch(name) is *NOT* on, then do consequences. Here's a CC syntax example:

EXAMPLE

```
       ")"  >  ifn(paren)  "EXTRA ) FOUND AT"
                           out(book,chpt,verse) nl
               endif
```

This `ifn` command can also be used with `else` and alternate consequences. However, there would be little reason not to use `if` rather than `ifn` when there are consequences for both states. Combining `if` and `ifn` conditions is acceptable. For example, `if(\s) ifn(\r)` would require '\s' to be *on and* '\r' to be *off* for the consequences to be performed.

3. Additional tips and techniques involving switches.

a. Setting/clearing a switch twice—There is no harm in setting (turning on) a switch that is already on, or clearing (turning off) a switch that is already off. This is sometimes deliberately done to ensure that a switch currently has the correct setting.

b. ??? Search Technique—We have a number of established CC tables which are used in the preprocessing stage of preparing a manuscript for publishing. Several of them offer one or more options that can be selected (See Figure 11.). An option is usually selected by setting a switch. The table is constructed with the `set` switch command already in the `begin` entry. This `set` command can be deactivated or reactivated by inserting or removing a `c` from the beginning of the line. (The `c` makes the line a comment so it is not performed.)

   In order to make the table more easily modifiable, the option switches (the `set` commands activating the options) are usually marked with ??? (triple question marks) in the comments at or preceding each such `set` command. In this way, the user can 'search' the table for ??? using his word processor and easily find the options and their `set` commands. (This ??? is also used to mark any parts of the table which may need job specific modification such as stores, etc.)

Fig. 11. FLAGEM.CCT

```
c           Mod 53        14-DEC-90   LIB:FLAGEM.CCT

c  WARNING: ERRONEOUS CHARACTERS that occur in id lines,
c    book titles, picture captions, & footnotes will not be
c    found by this table. Only those found in para-
c    graphs, poetry, or section heads will be found!

c  Search for "???" to locate modifyable sections of table.
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
begin       >                 clear(ENG,PORT,SPAN)
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? SELECT THE PREDOMINANT QUOTE SYSTEM USED:
c ENGLISH - the default:
              set(ENG)
c PORTUGUESE:
c             set(PORT)
c SPANISH:
c             set(SPAN)
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? IF THE PREDOMINANT QUOTE SYSTEM IS NOT ENGLISH
c           (QUOTE MARKS "<<" ARE NOT USED)
c                       OR
c    IF "<<" IS NOT REQUIRED WHEN A CITATION CONTINUES
c    AFTER A NEW PARAGRAPH, CHANGE THE FOLLOWING STATEMENT:
c ENGLISH style - the default:
              set(<<PAR)
c not ENGLISH style:
c             clear(<<PAR)
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? SELECT THE QUESTION-EXCLAMATION MARK SYSTEM USED:
c ENGLISH - the default:
              clear(SP??)
c SPANISH:
c             set(SP??)
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? SELECT THE CROSS REFERENCE STYLE OF THIS DOCUMENT:
c CROSS REFS IN PARENS - the default
            set(Refinpar)
c CROSS REFS WITHOUT PARENS
c           clear(Refinpar)
c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? TO ALTER THE BOOK AND CHAPTER DISPLAY:
c             DISPLAY ALLOWED - the default
            set(disp)
c             DISPLAY INHIBITED
c           clear(disp)

c ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c    ??? TO IGNORE FOOTNOTE MARKER REPORTING:
c             REPORT ON FOOTNOTE MARKERS - the default
            clear(ignfm)
c             IGNORE FOOTNOTE MARKERS:
c           set(ignfm)
```

c. Mark and Rewind—Sometimes it is necessary to make changes that depend on what follows a match. Mark and Rewind is a technique to get CC to look ahead in a file, see what is there, and return to the original match to make the proper change.

Let's say we want to put out one of two Ventura tags for a section head (\s) depending on whether the text element following it is a chapter (\c) or not. Figure 12. shows a technique for doing this. In this example, the text elements in question closely follow each other, so a store could have been used. But sometimes a great distance must be covered in the text file to find the needed information before making the current decision. This technique is most helpful then.

When a section head is encountered, "<MARK>" is inserted into the output file. This is to mark our current place in the input file. (We could have inserted "XXX" or any other text that we were certain did not otherwise exist in the input file.)

Then we look ahead, doing no other processing but looking for the next text element. All of the input text is passed unchanged to

Fig. 12. Mark and Rewind Technique

```
C EXLSN5B.CCT -- example of a table that looks ahead of a
C match to see what follows, and makes a different
C replacement depending on text that follows the match.

group(sfm)    C finds sfm's
    '\v '   > '@verse = ' C outputs verse paragraph tag
    '\s '   > '<MARK>'    C found sect hd, outputs a mark
              use(look_ahead) C look ahead to next sfm
    '\c '   > '@chapter = '   C output chpt paragraph tag

group(look_ahead)         C looks ahead to next sfm
    '\c '   > dup             C found chapter sfm, dup
              set(chapter)    C set chapter switch
              use(rewind)     C go back to mark
    '\'     > dup             C found sfm other than chpt
              clear(chapter)  C clear chapter switch
              use(rewind)     C go back to mark

group(rewind)          C find mark, output proper para tag
    '<MARK>'  > if(chapter)   C found mark, if chpt sw set,
              '@Chap. Section = '   C output chpt sect tag
              else            C else, chpt switch not set
              '@Reg. Section = '    C output reg sect tag
              endif
              use(sfm)        C return to sfm search
    ''      > back(1)         C backup till <MARK> found.
                              C (max of 200)
```

the output file until the next element is encountered. If it is a reference (\c), a switch is set. If it is any other element (\), the switch is not set.

Now we begin the 'rewind' process, backing up one character at a time—taking it out of the output and putting it back into the input. Each time we put a character back into the input we check again to see if the next six characters in the input are `"<MARK>"`. Eventually, when we have backed up enough characters, we will have backed up over our `"<MARK>"`, putting it back into the input. At this point we have a match and we know we are back where we started from—only this time our switch will tell us whether the following element is a chapter or not.

PRACTICE ACTIVITIES and QUESTIONS

1. `List FLAGEM.CCT` and search for the switch named 'SecHd'. For each occurrence, list below the search side of the entry and the switch command used (`set`, `clear`, `if`, or `ifn`).

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

2. Starting at the beginning of FLAGEM.CCT, locate all '???'s.
   How many are there? _____
   How many are for setting or clearing switches? _____

List the names of the switches at these locations.

_____

_____

_____

_____

3. There are some very strange looking switch names in this table. In the section of `group(1)` dealing with quote marks, there is a replacement action containing:

```
ifn(<<-<-<<,<<-<,<<,<) ...
```

   a. how many switches are named? _____
   b. what does it mean, in terms of switches being on or off?

_____

_____

READING ASSIGNMENT

CC User's Guide: 14 clear, **19** ifn

EXERCISES

1. Copy FIXEM.CCT to a file named MYFIXEM.CCT. Search for '???' and make any alterations necessary to:
   a. allow display;
   b. alter line length to '70' characters;
   c. strip footnotes;
   d. not insert footnote markers;
   e. strip illustrations;
   f. allow only ',' and '–' with verse numbers;
   g. allow '\eq' to be a legitimate SFM, in `group(2)`;
   h. in `group(10)`, enable the changes '-u' to '_u'; '-U' to '_U'; '`"`' to '/'; and '`::`' to '`:`'; and add the change '=' to '`-`';
   i. in the two entries with the `XXX` for the backup and rewind, change the `"XXX"` to `"&&&"`.

   Run CC in display mode using SOMT.SFM as input and observe the changes, especially from the insertion of the `"&&&"` through its removal. Print your output file and hand in.

Mod 12 'If' Commands Using Stores

COMMANDS/TOPICS COVERED

ifeq	**ifneq**	**cont**
set(dummy)	**ifgt**	**incr**

OBJECTIVES

At the end of this module, the student will be able to:

- modify and/or write tables correctly using `cont`, `ifeq`, `ifgt`, `ifneq`, and `incr`;
- demonstrate an understanding of counter initializing by writing the output for specific input text and table entries using counters.

INSTRUCTION

1. **Comparing storage areas to strings using** `ifeq(name)`— A replacement action can be dependent on the contents of a *storage area*. In this command, if the character string stored in a storage area 'name' equals another designated string, then the consequences are performed.

   EXAMPLE

   ```
   "\s"  >  ifeq(lastSFM) "\s" out(book,chpt)
                                "Successive Sect.Hds Found"
            endif
   ```

   The replacement side would read: if the string stored in the storage area named 'lastSFM' equals '\s', then send the contents of the storage areas named 'book' and 'chpt' to the output, followed by the string 'Successive Sect.Hds Found'.

   The `ifeq` command is a type of 'if' command that requires a storage area name and a character string, and is similar to `if` in that it may also use `else`, alternate consequences, and `endif`.

   The character string which the storage area is compared to may include a character string within delimiters (as shown in the above example), `nl`, and ASCII numbers. The string is terminated by the next *command* (except for `nl` or `c`). In the above example, the string is terminated by the `out` command.

   But what if we wanted to put out the message within the delimiters *first* and then the book and chapter from the storage areas?

BAD EXAMPLE

```
"\s"  > ifeq(lastSFM) "\s"  "Successive Sect.Hds Found at"
                              out(book,chpt)
        endif
```

In this example the CC program would have considered the *entire* string `"\sSuccessive Sect.Hds Found at"` to be the string that storage area 'lastSFM' must be compared to—with the consequences being *only* `out(book,chpt)`! How could we avoid this? There must be an intervening command between the two strings. One possibility is to set a 'dummy' switch which serves *no* purpose except to exist there as a *command*, thereby signaling the end of the string for the compare:

EXAMPLE

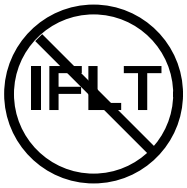
```
"\s"  > ifeq(lastSFM) "\s"  set(dummy)
                           "Successive Sect.Hds Found at"
                           out(book,chpt)
```

Now, the `set(dummy)` command will terminate the string that 'lastSFM' must equal, and the string 'Successive Sect.Hds…' is recognized as a consequence.

2. **If not equal command** `ifneq(name)`— This command works exactly the same as the `ifeq` command except that it means "if the character string stored in the storage area 'name' is *NOT* equal to…"

3. **If greater than command** `ifgt(name)`— This related command is most commonly used to compare numeric strings with each other. It means "if the number in storage area 'name' *is greater than* the number designated in the numeric string following it, then perform the consequences".

   NOTE: There is *NO* `iflt` or 'if less than' command!



   We will cover two more commands before looking at some more examples.

4. **The contents command** `cont(name)`— This means "the contents of the named storage area"—that's all! It can be used in conjunction with one of the above 'if' commands—replacing the *string* to be compared with:

EXAMPLE

```
"\s"  >  store(newSFM) dup endstore
         ifeq(lastSFM)  cont(newSFM)
                                "Successive \s found at"
                                out(book,chpt)
```

Note that since `cont` is used instead of a string, the dummy switch is *not* needed. There is no string to terminate.

The command `cont` may be used in other types of entries as well—nearly any place a string could be used. It can even be used as a search argument:

EXAMPLE

```
cont(lastSFM) > ifn(verse) "Successive"  out(lastSFM)
                           "Found at"  out(book,chpt)
                endif
```

5. **Incrementing a storage area using** `incr(name)`— The command `incr` will cause the value stored in the named storage area to increase by one. This is a useful command for counting the number of times an item is found, or for counting the times an event occurs (such as how often a replacement action has been performed).

EXAMPLE 1

```
begin   >  store(\s) "0" endstore
"\s"    >  incr(\s)
endfile >  out(\s)  " section heads were found"
           endfile
```

EXAMPLE 2

```
"\"     >  "Unidentified SFM found at"
           out(book,chpt)
           incr(errcount)
           ifgt(errcount) "5"  set(dummy)
              "Text not ready for processing" nl
              "At least " out(errcount) " errors found."
              endfile
           endif
```

In these examples, `incr` is used to increment storage areas which are being used as *counters*. In the event a storage area is not initialized (i.e., created using `store` and the string ‘0’ placed in it) and it is incremented for the first time using `incr`, then the program will first create such a storage area, pretend it contains a ‘0’, then increment it to ‘1’.

Notice what would happen in Example 1 if the storage area ‘\s’ were not initialized to zero (or not created) and if no section heads (\s) were encountered. The `out(\s)` command would then output a storage area without anything in it; ‘nothing’—*a null* rather than a

zero—would have been sent to the output at the end of file. The message will then read ' *section heads were found'* rather than *'0 section heads were found'*!

If it will minimize confusion and aid readability and understandability, the storage areas should be initialized in the `begin` statement. This can be used as documentation of all the stores which are used in the table and, with comments, can explain the use of each.

Several examples are included below which use the commands discussed in this module. They are portions or complete tables which are used in the preprocessing stage of manuscript publishing. The Practice Activities and Questions which follow the examples will make use of them.

EXAMPLES

Fig. 13. from VPNAM.CCT

```
c    VPNAM.CCT             Mod 89      31-JAN-91
c    from VPNAM.CCT        Mod 87      15-JAN-91
c           modified for PAD-CCP, May 1991, by K. Seitz
c  A table to convert SFMs to VP tags for Scripture
c  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     ...
   c Outputs an incremented letter for the footnote marker.
group(105)
     any(Num)     > dup
     'a*'         > endstore  c "a*" indicated multiple
                    ifeq(mkrlet) 172
                              c If the last letter was z,
                      store(mkrlet) 141 endstore
                              c start over again with a.
                    else incr(mkrlet)
                    endif
                    out(mkrlet) back(1) use(106)
     '*'          > endstore
                    ifeq(mkrlet) 172
                              c If the last letter was z,
                      store(mkrlet) 141 endstore
                              c start over again with a.
                    else incr(mkrlet)
                    endif
                    out(mkrlet) back(1) use(106)
     any(Div) '*' > endstore  c The letter is not
                    out(mkrlet) back(1) use(106)
                              c  other than the 1st one
                              c  for this footnote.
```

Fig. 14. LONGWD.CCT

```
C ********************************************************

C LONGWD.CCT modified for PAD-CCP, May 1991, by K. Seitz
C -- a table to make a list of all words of certain length
C    or longer.
C           For use in setting up a hyphenation table.

C ********************************************************

begin       > store(1) " " nl ",.:;><!\|()$#/'0123456789"
              endstore
              store(2) '6' endstore
              store(3) '0' endstore
               store(5) 'abcdefghijklmnopqrstuvwxyz'
               'ABCDEFGHIJKLMNOPQRSTUVWXYZ' endstore

group(1)
    any(5)    > store(4) dup incr(3) use(10)
    any(1)    > ''
    ''        > '' fwd(1)
    endfile   > endfile

group(10)
    ''        > dup fwd(1) incr(3)
                ifeq(3) cont(2) use(20) endif
    any(1)    > store(3) '0' endstore use(1)

group(20)
    ''        > dup fwd(1)
    any(1)    > endstore out(4) nl
                store(3) '0' endstore use(1)
```

Fig. 15. from FIXEM.CCT

```
c           FIXEM.CCT  Mod 24       16-JAN-91

    ...

    c ccccccccccccccccccccccccccccccccccccccccccccccc
    c ???LIST job specific orthographic corrections!
    c       These are shown as examples only.
group(10)
c   'c'     > '/o' incr(char,char)    c au
c   'C'     > '/O' incr(char,char)    c AU
c   'j'     > '/e' incr(char,char)    c epsilon
c   'J'     > '/E' incr(char,char)    c EPSILON
c   'q'     > '/n' incr(char,char)    c eng
c   'Q'     > '/N' incr(char,char)    c ENG
c   '-u'    > '_u' incr(char,char)    c barred u
c   '-U'    > '_U' incr(char,char)    c BARRED U
c   '"'     > "/"  incr(char)         c Glottal

c   '['     > '((' incr(char,char)    c Open bracket
c   ']'     > '))' incr(char,char)    c Closing bracket
c   '"'     > "'"  incr(char)         c Grave accent
c   '::'    > ':'  incr(char)         c colon
```

Fig. 16. from CSTCHK.CCT

```
c  CSTCHK.CCT     CHARACTER SPECIFICATION TABLE CHECKER
c                  28-NOV-88     KH

c  CSTCHK.CCT - a validity check for Character Specifica-
c  tion Tables that will find certain errors that cannot
c  be found by the CST compiler

            ...
group(25)             c  store octal access code
  '/' any(10) > '/' endstore set(2) use(45)
                        c  composite   character follows
  any(15) > dup incr(3)   c  valid octal character
  any(10) > endstore      c  white space ends access code
            ifeq(3) '4'
              ifgt(1) '0377'  c  compare 4-digit num.
                        set(18)
            endif             c  invalid access code
            ifeq(3) '3'
              ifgt(1) '377'   c  compare 3-digit number
                        set(18)
            endif             c  invalid access code
            use(45)
  '/*'    > set(15) ' /*' back(3)
                    c  white space required before comment
  endfile > do(3) set(14) use(99)   c  no SILID found
  ''      > set(18) use(20)         c  invalid access code

group(30)               c  store decimal access code
                           (decimal point is valid)
  './' any(10) > '/' endstore c composite char. follows
                 set(2) use(45)
  '/' any(10)  > '/' endstore c  composite char. follows
                  set(2) use(45)
  any(16)      > dup incr(3)  c  valid decimal character
  '.' any(10)  > endstore use(45)
                        c  decimal & white space ends access
  any(10)      > endstore c  white space ends access code
                 ifeq(3) '4'
                   ifgt(1) '0256' c  compare 4-digit number
                       set(18)    c  invalid access code
                 endif
                 ifeq(3) '3'
                   ifgt(1) '256'  c  compare 3-digit number
                       set(18)    c  invalid access code
                 endif
                 use(45)
  '/*'    > set(15) ' /*' back(3)
                     c  white space required before comment
  endfile > do(3) set(14) use(99) c  no SILID found
  ''      > set(18) use(20)       c  invalid access code
```

Fig. 17. from WRDLST.CCT

```
c    WRDLST.CCT        Mod 2
c This is a table to remove all data from word list files
c produced by TAD so each word is listed only once and
c placed on a new line without its count or references.
c storage (let) must contain all characters found in words
c in word list

begin       >
              store(lastwd)   ''    c Stores previous word
              store(thiswd)   ''    c Stores current word
              store(num)  '1234567890' c numbers for count
              store(let)  "abcdefghijklmnopqrstuvwxyz'"
                          "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
              endstore  use(1)

    ...

    c ccccccccccccccccccccccccccccccccccccccccccccccccc
    c AFTER the entire word has been stored,
    c compare it with the previous word.
group(5)
  10        > ''              c Strip backspace commands
  any(let)  > dup
  ' '       > endstore
              ifeq(lastwd) cont(thiswd)c This word is the
                 use(6)                c  same as the
              endif                    c  previous word.
              ifneq(lastwd) cont(thiswd) c This word is not
              out(thiswd) store(lastwd) c  the same, output
              outs(thiswd) endstore    c  it to the file
                 use(6) endif          c  and to the
                                       c  previous word
                                       c  storage area.
```

PRACTICE ACTIVITIES and QUESTIONS

1. Explain what is happening in the `ifeq` through `endif` commands in Fig. 13 (VPNAM.CCT, `group(105)`).

   _____

   _____

   _____

   _____

   _____

2. In Fig. 14 (LONGWD.CCT), `incr(3)` means to add 3 to the counter.

   ☐ True ☐ False

3. Write out the meaning of the `ifeq` through `endif` commands in Fig. 14.

_____

_____

_____

4. In Fig. 15 (FIXEM.CCT, `group(10)`), why is the counter name repeated in the `incr` command in several entries?

_____

_____

_____

5. In Fig. 16 (CSTCHK.CCT, `group(25)` and `(30)`), write out the interpretation of: `ifgt(1) '377' set(18)`

_____

_____

_____

6. Study the `ifeq` and `ifneq` entries in Fig. 17 (WRDLST.CCT) and rewrite that whole replacement entry to simplify it.

_____

_____

_____

_____

_____

_____

_____

7. Would it be necessary to intialize the counters in the following situations?

a.
```
endfile > ifgt(errcnt) '0'
                  out(errcnt) ' errors found in file'
           else   'no errors in file'
          endif
          endfile
```
☐ Yes ☐ No

b.
```
endfile > out(errcnt) ' errors found in file'
          endfile
```
☐ Yes ☐ No

READING ASSIGNMENT

CC User's Guide: 14 cont, **18** ifeq, **19** ifgt, **19** ifneq, **20** incr

EXERCISES

1. There seems to be a slight ambiguity between the comments at the beginning of 7CHAR.CCT and the comments in `group(2)` where the word length is tested. (See Fig. 7 in Mod 4.) By looking at the actual commands, determine whether words exactly 7 characters in length will be deleted or sent to output.

   ☐ 7 character words will be deleted.

   ☐ 7 character words will go to output.

2. Copy this file from your disk to a file named MYCHAR.CCT, and modify it as follows:
   a. In the `begin` entry, provide a store marked by ??? for the user to indicate the maximum length of words to be deleted;
   b. use this store in `group(2)` in testing the word lengths;
   c. complete the cleanup by appropriately changing comments and deleting the ??? in `group(2)`.

   Run CC, using first the *original* table and then your modified table. Use SOMT.WDL as input. Compare the results—they should be the same. Print your table; *write your answer to Exercise 1 on the bottom*; and hand it in.

Mod 13 'If' Commands—Advanced Techniques

COMMANDS/TOPICS COVERED

begin	**more mark and rewind**
end	**nesting 'ifs'**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of nesting 'if' commands by modifying and/or writing tables using nested 'ifs', at least two deep, and properly using `begin` and `end`.

INSTRUCTIONS

1. **Variation on Mark and Rewind Technique—** In Mod 11, a technique called 'mark and rewind' was described. It enabled the program to look ahead in order to base a current decision on future text. In another variation of this, a *current condition* may require a change in text *already* processed.

EXAMPLE (taken from FIXEM.CCT group(10), (50) and (51))

```
...
group(10)          c OUTPUT and COUNT this character.
          c If the current character would make the
          c  line too long, 'XXX' is inserted to
          c  mark the place in the word.
  ""    >  fwd incr(char)
           ifgt(char) cont(maxchars) "XXX"
                                    back(4) use(50)
           endif

group(50)          c BACKS UP to the previous space and
                   c  ends the line.
  " "   >  nl store(char) "00" endstore
           use(51)
  ""    >  back(1)

group(51)          c GOES forward to 'XXX' to prevent
                   c  rechanging characters
  "XXX" >  "" use(1,10)
  ""    >  fwd incr(char)
```

In the above example, the length of the line is being limited to a maximum number of characters. This number has been stored in 'maxchars' storage area. If a character being forwarded to output causes the character count for that line to exceed the maximum char-

acter limit, then 'XXX' is written to output to mark the current position. Then the XXX and the last character of output are brought back from the output and put in the input. A search is begun for the most recent space in the output—backing up one character at a time.

When that space is found, the line is terminated (`nl`), the line character counter is zeroed, and the search is begun for the forward point of processing marked with XXX. Upon finding this mark, normal processing is resumed.

This example and the one in Mod 11 show that similar techniques can be used to either look *ahead* or *behind* of the current place and then return.

2. **Nesting 'if' commands using `begin` and `end`—** Sometimes the actions to be taken depend on complex conditions. For example, if the light is on in the room *and* if the time is between 7 p.m. and 10 p.m., then someone is in there—go in and visit him. But if the light is on *and* it is later than 10 p.m., someone forgot to turn the light out—so turn off the light. If the light was not on, then no one is there, so go watch TV. We might view this as follows:

```
if(light)
          ifgt(time) "1859"
                         ifgt(time) "2200" clear(light)
                         else do(visit)
                         endif
          endif
else
use(TV)
```

Confusing, isn't it?! Nesting 'if' commands can be complex and difficult to follow. Some may have `else` and alternate consequences associated; others may not. By paying attention to alignment, we can make the table a bit more readable for us humans, but unfortunately the program *does not read* alignment. In the pseudo example above, something more is needed to make the program properly do what we *intended* to say, not what in reality we said.

`Begin` and `end`—You are now familiar with using `begin` as the first entry on the *search* side. But it is also used on the *replacement* side to mark the beginning of a set of actions which are terminated with `end`. These commands (`begin` and `end`) are necessary when nesting 'if' commands. Our pseudo example might now be written:

```
if(light)
   begin
      ifgt(time) "1859"
         begin
            ifgt(time) "2200"
               clear(light)
            else
               do(visit)
            endif
         end
      endif
   end
else
   use(TV)
endif
```

This may look more confusing than it is. Here's the rule for each level of 'if':

```
if condition
   begin
      actions (including paired if/endif commands)
   end
else
   begin
      actions (including paired if/endif commands)
   end
endif
```

When the 'actions' contain additional 'ifs' the same rule applies. When alternative consequences do not apply to the 'if' condition, the `else` and its associated `begin`...`end` may be omitted.

There is a simple test you can perform to determine if your table is nested properly. Draw a line from the 'i' of every `if` command to the 'e' of each respective `endif`, then likewise from the 'b' of every `begin` to the 'e' of each respective `end`. When an `if/endif` pair contains an `else`, draw the line so that it touches the 'e' of `else`. Lines should be straight, and must be drawn to the left of all other commands. If your lines can be drawn *without crossing each other*, and the `if/endif` lines are *always separated by at least one* `begin/end` *line*, then the change table is nested properly.

There are endless variations of nesting. Figures 18, 19, and 20 show a number of examples.

VOCABULARY and CONCEPTS

nesting—this is a computer programming term. When referring to 'if' commands, it means an 'if' command contained within the replacement actions of another 'if' command.

Fig. 18. from VPNAM.CCT group(3)

```
          c COMPLETION OF VERSE NUMBERS OTHER THAN VERSE 1
group(3)
  "-"      > append(VsNum) outs(NBHy)   c hyph in bridge
  any(Num) > dup                        c numbers
  any(sp)  > '' endstore do(2) do(3)    c space or new line
             if(VInPro)                 c mid-para vs num
               begin
                 if(VsSty) begin        c in verse style
                             do(10) '@VRS PAR = '
                           end
                 endif
                 if(ParSty) begin       c in par style
                              if([)         c no sp tween
                                 begin      c [ & num
                                   ''
                                 end
                              else
                                 begin
                                   ' '      c space
                                 end
                              endif
                            end
                 endif
               end
             endif
             use(1,10)
```

Fig. 19. from VPNAM.CCT group(107)

```
 c Completes the reference marker at the beginning of
 c   a footnote. May convert the sequential number to
 c   a sequential letter (a thru z, lower case only).
group(107)
  any(Num) > dup
  any(sp)  > '' endstore
             if(FMkrL) begin          c Marker is a ltr,
                         ifeq(fnlet) 172    c Reset to a
                           store(fnlet) 141 c if z was the
                           endstore         c last letter.
                         else         c Else,
                           incr(fnlet)      c change to
                         endif              c  next letter.
                         out(fnlet)   c Ltr, .5 thin sp
                       end
             endif
             if(FMkrN)                c Marker is a number
                  out(tempn)          c Output the number
             endif
             if(FMkrS) begin          c Same mrkr in text.
                         if(FMkr) out(FMkr) c Mrkr wanted.
                         else         c No marker wanted.
                         endif
                       end
             endif
             store(tempr) use(108)
```

Fig. 20. from STDFIX.CCT group(10)

```
nl "\v" > ifn(\p\q\m)
            begin
              if(\s) begin                    c see note*
                       nl "\m "
                       clear(\e)
                     end                      c see note*
              else
                     begin
                       if(\c) begin           c see note*
                                nl "\m "
                              end             c see note*
                       endif
                     end
              endif
            end
          endif
```

* the table will execute correctly without these two `begin/end` sets since the actions do not contain `if` commands. All other `begin/end` sets are essential.

PRACTICE ACTIVITIES and QUESTIONS

1. Take time to study Figures 18, 19, and 20. Identify the commands being used; whether they deal with switches, stores, groups, or other; and how the logic flows.

2. Look at the use of `begin` and `end` in Figures 18, 19, and 20. Show that these tables are properly nested by drawing lines connecting all `begin` and `end` commands and also `if`, `else`, and `endif` commands. Notice that successive ‘if’ commands do not require `begin/end` when the ‘if’ is completed (with `endif`) before the next ‘if’ begins. But ‘ifs’ which are nested (one not completed before the next begins) *do* require a `begin` and `end` for each ‘if’. Fig. 19 contains both successive and nested ‘ifs’. Can you identify each?

READING ASSIGNMENT

CC User’s Guide: 12 begin, **16** end

EXERCISES

1. `List WDLENG.CCT`, paying special attention to the `store` commands in the `begin` entry and the replacement side for `any(sp)` in `group(2)`.
   – Make a copy of this file called MYWDLENG.CCT.

- Modify it so that after the letters of each word are counted, the word lengths are tabulated by ranges as follows:
  1–6 characters
  7–15 characters
  16–24 characters
  over 24 characters
  (Use the `ifgt` command for this exercise. Do not test for each word length individually.)
- The statistics are put out at the end of file in `define(5)`. You need not understand defines, only that they are a list of replacement actions. Modify these appropriately.
- Run `CC`, using first the *original* table and then your modified table. Use SOMT.WDL as input. You should be able to reconcile the two sets of statistics.
- Print your table and hand it in.

2. You are processing a Scripture file containing clean text using only five SFMs. Write a table to convert the SFMs to the appropriate Ventura tag name according to the chart below. Assume that each SFM starts on a new line and that a single space follows it.

SFM	Tag Name	Condition
\mt	@TITLE M =	always
\c	@CHP =	always
\s	@SEC CH REF =	when at a chapter break and followed by a cross ref.
	@SEC CH =	when at a chapter break, no cross ref.
	@SEC REF =	when followed by cross ref, but no chapter break
	@SEC =	when no chapter break, no cross ref.
\r	@REF =	always
\p	@PAR 1ST =	when first paragraph after a chapter break
	@PAR =	all other paragraphs

No matter which order the \c and \s are in, output them in the following order: \s, \r, \c, \p (for whichever elements are present). This order will be needed for dropped chapter numbers. Use MOD13.SFM on your disk for input. Print your table and hand it in.

Hint: Try setting a switch and storing chapter, section, and reference when they are encountered. When a \p is found, then test for what preceding elements were found and stored, and put them out in the proper order with appropriate tags. (Assume there will only be one SFM of a kind prior to a \p.)

Mod 14 'Doing' Defined Routines and Repeating

COMMANDS/TOPICS COVERED

define **do** **repeat**

OBJECTIVES

At the end of this module, the student will be able to:

- draw a block diagram of a table containing a define/do operation;
- modify and/or write tables properly using `define`, `do`, and `repeat`.

INSTRUCTION

1. **A shortcut for repetitive actions using** `do(name)` **and** `define(name)`—
When the same set of replacement actions must be performed at multiple locations within a change table, the table can become cumbersome and unnecessarily long. (See Fig. 21.)

   The same results can be accomplished by extracting these identical routines and defining them with the `define` command. On the search side the entry would be:

   ```
   define(name)      >
   ```

   with `name` being whatever you choose to call it—following the same rules that apply to groups, stores, and switches. The replacement side would contain the set of actions to be performed.

   In the replacement arguments from which these defined actions were removed, insert:

   ```
                     >     do(name)
   ```

   naming the specific `define` containing the actions to be performed at this point in the processing. (See Fig. 22.)

   Other replacement actions can precede and/or follow the `do` command. After the defined routine is performed, the program will return to where the `do` command was located, and the table processing will continue from there.

   All of the `define` commands and their replacement actions should be sequenced together in the table—either between the `begin` search command and the first `group`, or after the last `group`. If the `define`

Fig. 21. from EXTRAC.CCT

```
c    EXTRAC.CCT            Mod 2       13-JUN-89
c    Use: for extracting selected SFM elements
c    Based on FLAGEM.CCT
c    Search for "???" to find sections of the change table
c    that you must alter.
begin       >
         store(sp) ' ' nl            C SFM terminators
         store(BK) ''          C for Book name storage
         store(CH) '0'         C Chapter number storage
         store(VS) '0'         C Verse number storage
         store(num) '1234567890'       C Legit chapter num's
         store(VsDiv) 'abc'            C verse divisions
         store(BkMsg) nl 'BOOK:  '     C book name message
         store(ChMsg) 'Chap:  '        C chapter message
         use(1)

            c ccccccccccccccccccccccccccccccccccccccccccc
            c Finds sfm initiators and strips all else.
group(1)
     '\'     > use(2)
     ''      > omit
     endfile > endfile

            c ccccccccccccccccccccccccccccccccccccccccccc
            c Identify required sfms to be extracted.
            c Also store book, chapter, and verse for
            c   reporting (if required).
group(2)
                      c ???Insert required sfms here
c    '???' any(sp) > next
c    '???'         > out(BK,CH,VS) 11     c Output found SFM.
                     ifn(3nums) 11 endif
                     '\' dup use(10)

c    '???' any(sp) > next
c    '???'         > out(BK,CH,VS) 11     c Output found SFM.
                     ifn(3nums) 11 endif
                     '\' dup use(10)

c    '???' any(sp) > next
c    '???'         > out(BK,CH,VS) 11     c Output found SFM.
                     ifn(3nums) 11 endif
                     '\' dup use(10)

                      c If \s is extracted, protect \st!
     'st'         > omit use(1)
     's' any(sp)  > next
     's'          > out(BK,CH,VS) 11
                    ifn(3nums) 11 endif
                    '\' dup use(10)

                      c Store book, chapter, verse number.
     'c' any(sp) > next    c Stores chapter number.
     'c'         > store(VS) '0' endstore c Zero out vs num.
                   clear(CH,VS,3nums)
                   store(CH) use(3) c Begin storing chp num.
```

Fig. 22. EXTRAC.CCT

```
c    EXTRAC.CCT           Mod 2       13-JUN-89
c    Use: for extracting selected SFM elements
c    Based on FLAGEM.CCT
c    Search for "???" to find sections of the change table
c    that you must alter.
begin       >
         store(sp) ' ' nl     C SFM terminators
         store(BK) ''         C for Book name storage
         store(CH) '0'        C Chapter number storage
         store(VS) '0'        C Verse number storage
         store(num) '1234567890'     C Legit chapter num's
         store(VsDiv) 'abc'          C verse divisions
         store(BkMsg) nl 'BOOK:  '   C book name message
         store(ChMsg) 'Chap:  '      C chapter message
         use(1)
                        c Outputs extracted element.
define(Extract)         >
              out(BK,CH,VS) 11
              ifn(3nums) 11 endif
              '\' dup use(10)

            c ccccccccccccccccccccccccccccccccccccccccc
            c Finds sfm initiators and strips all else.
group(1)
     '\'     > use(2)
     ''      > omit
     endfile > endfile

            c ccccccccccccccccccccccccccccccccccccccccc
            c Identify required sfms to be extracted.
            c Also store book, chapter, and verse for
            c   reporting (if required).
group(2)
                        c ???Insert required sfms here
c    '???' any(sp) > next
c    '???'         > do(Extract)     c Output found SFM.
c    '???' any(sp) > next
c    '???'         > do(Extract)     c Output found SFM.
c    '???' any(sp) > next
c    '???'         > do(Extract)     c Output found SFM.
                        c If \s is extracted, protect \st!
     'st'       > omit use(1)
     's' any(sp) > next
     's'         > do(Extract)
                        c Store book, chapter, verse number.
     'c' any(sp) > next            c Stores chapter number.
     'c'         > store(VS) '0' endstore c Zero out vs num.
                   clear(CH,VS,3nums)
                   store(CH) use(3) c Begin storing chp num.
```

(continued on next page)

Fig. 22. EXTRAC.CCT continued

```
    'id' any(sp) > next
    'id'         > ''                 c Strip "\id"
                   store(BK) fwd(3)   c Stores book name.
                   endstore
                   wrstore(BkMsg)     c Display book name.
                   wrstore(BK) write nl
                   out(BK)            c 1st 3 ltrs of name
                   append(BK) ' ' endstore c Space after bk
                   store(CH,VS) '0' endstore
                   use(10)            c Output id line.

    'v' any(sp) > next                c Stores verse number
    'v'          > clear(VS) store(VS) use(4)
    ''           > omit use(1)        c SFM not requested

           c ccccccccccccccccccccccccccccccccccccccccccc
           c Finish storing the chapter number.
group(3)
    any(num) > dup                     c Dup numbers.
               if(CH) set(3nums) endif
               set(CH)
    any(sp)  > endstore wrstore(ChMsg) c Display current
               wrstore(CH) write nl    c  chapter number.
               append(CH) ':' endstore c Add a colon after
                                       c  chapter number.
               '' use(1)

           c ccccccccccccccccccccccccccccccccccccccccccc
           c Finish storing the verse number.
group(4)
    any(VsDiv) > dup          c Verse division!
    ','        > next         c Verse bridge.
    '-'        > dup
    any(num)    > dup
                  if(VS,CH) set(3nums)
                  endif
                  set(CH,VS)
    any(sp)    > endstore '' use(1)

           c ccccccccccccccccccccccccccccccccccccccccccc
           c Output the requested element.
group(10)
    any(sp) '\' > next
    '\'          > nl use(2)
```

routines are short and aid significantly in understanding what the table is about, it would be best to put them after the `begin` entry. If they are lengthy, mundane routines such as report writing, they would best be placed at the end. *Under no condition* can they be placed before the `begin` entry.

Sometimes when the `begin` entry has lengthy commands that do not

add to the understanding of the table, these commands can be placed in a `define`. The `define` can then be placed at the end of the table where it does not hinder the readability of the table. This is done even when the routine will only be performed once.

For consistency, place all `define` routines together in the table.

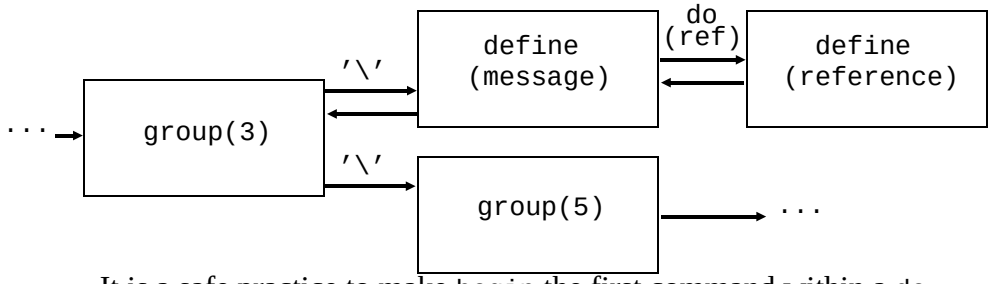
`Do/define` routines can also be nested to a depth of 10. Nesting refers to cases where a `define` contains within its replacement a subsequent `do` command, as shown in the following example:

EXAMPLE

```
...
group(3)
nl "\"  > next
"\"     > endstore    c stop store at end of last element
          do(message)
          nl "\" back(1)
          use(5)
...
define(message)    >  do(reference)
                      out(capture)
define(reference)  >  out(book, chpt, verse)
                      " contained "
```

When the input matches one of the search entries in `group(3)`, the current storing will be ended and the actions defined as ‘message’ will be performed. The first action in ‘message’ calls for executing the actions in ‘reference’, so processing branches to this `define`. Here the contents of ‘book’, ‘chpt’, and ‘verse’ will be sent to output, followed by the sequence `" contained "`. Processing returns to the second line of `define(message)` and the string stored in storage area ‘capture’ is written to output. Then processing returns to `group(3)` and continues with outputting `nl` and the backslash, backs up over the backslash, and changes to `group(5)`.

This can be represented by the following block diagram.



It is a safe practice to make `begin` the first command within a `define` and make `end` the last, especially when its corresponding `do` command appears in the conditional part of any ‘if’ command.

2. **Re-executing a series of replacement commands with** `repeat`— This replacement side command causes the processing flow to go back to the previous `begin` in the replacement actions for that search entry:

EXAMPLE

```
"\"  >  do(errmsg)
        store(linechars) '0' endstore
        begin            c beginning of repeat loop
           "*" incr(linechars)   c fill out line with *
           ifneq(linechars) "68" repeat
                            c repeat until line filled
           endif
        end
        use(5)
```

In this example, the replacement actions would be executed in a sequential manner until the `if` command is encountered. At the `if` command, as long as the 'linechars' storage area contains a number not equal to '68', the `repeat` is executed, and the processing flow will immediately jump back to the previous `begin` and resume sequential execution. When '68' is reached in the 'linechars' counter, then the `repeat` command will not be executed and processing flow will continue on sequentially.

PRACTICE ACTIVITIES and QUESTIONS

1. `LIST` the following tables, and answer the associated questions for each:
   a. SEQ.CCT
      – How many times is `define(1)` executed? _____
      – Why was it made into a `define`?_____

      _____

      – When is `define(99)` executed? _____
      – What does it do? _____

      _____

   b. FLAGEM.CCT
      – What `define`s are used; how many table entries cause each to be performed; and what is the purpose of each?

Define	# of do's	Purpose of define
_____	_____	_____
_____	_____	_____
_____	_____	_____

c. CSTCHK.CCT

- Are there any nested "do's"? _____
- To what depth?_____
- `Define(2)` is only one line long. Is that a worth while use of define, and why? _____

_____

2. Fig. 23 shows the relevant portions of WDLENG.CCT. Rewrite the table (except for comments), placing the initializing of the counters into a `define`.

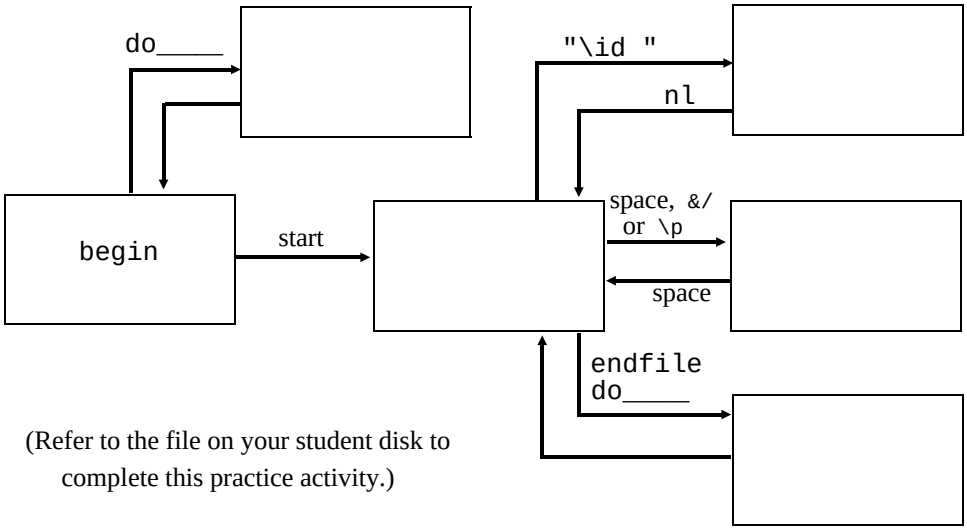
Fig. 23. from WDLENG.CCT

```
c    WDLENG.CCT         Mod 2        15-JUN-90
c    This table counts letters in words and outputs number
c    of occurrences of words with 1 character, 2 characters
c    etc found in the output of the word list program.
c    Assumptions:
c     1 All references have been deleted from the file .
c     2  "  page headings    "      "      "   "    "
c     3  "  reference counts "      "      "   "    "
c     4 An id line has been inserted in the file being read.
c     5 A \p occurs after id line and before first word.
c
begin   >   caseless
                       C COUNTERS FOR WORD LENGTH
       store(1)  '0'   store(2)  '0'  store(3)  '0'
       store(4)  '0'   store(5)  '0'  store(6)  '0'
       store(7)  '0'   store(8)  '0'  store(9)  '0'
       store(10) '0'   store(11) '0'  store(12) '0'
       store(13) '0'   store(14) '0'  store(15) '0'
       store(16) '0'   store(17) '0'  store(18) '0'
       store(19) '0'   store(20) '0'  store(21) '0'
       store(22) '0'   store(23) '0'  store(24) '0'
       store(25) '0'

       store(long) '0'     c Cntr for wds over 26 chars
       store(wds)  '0'     c Total word counter

       store(diac) "'~_" '"'   c diacritics
       store(sp)   ' ' nl  c word enders
       store(ct)   '0'     c counts characters
       endstore clear(1) use(1)
...
```

3. Below is a block diagram of WDLENG.CCT after being modified in Activity 2 above. Label the empty boxes and the blank underscore with the appropriate names of groups and defines.



(Refer to the file on your student disk to complete this practice activity.)

READING ASSIGNMENT

CC User's Guide: 14 define, **14** do, **23** repeat

EXERCISES

1. On a separate sheet of paper to turn in, draw a block diagram of 7CHAR.CCT, including its `define`. (Refer to Fig. 7 in Mod 4)

2. Fig. 24 shows an extract of table code. On the same paper used for Exercise 1, write the entries to consolidate the common commands to a `define`. Write both the `define` entries and the entries from which those commands were taken.

Fig. 24. from VPNAM.CCT

```
       c Non indented prose in an introduction.
'\im' any(sp,sp) > next
'\im' any(sp)    > next
'\im'            > if(2ndTag) nl nl
                       else set(2ndTag)
                   endif

       c Outline elements in an introduction.
'\io '  > if(2ndTag) nl nl
             else set(2ndTag)
           endif
           '@OUT INT = ' set(txt) use(1,10)
'\io1 ' > if(2ndTag) nl nl
             else set(2ndTag)
           endif
           '@OUT INT 1 = ' set(txt) use(1,10)
'\io2 ' > if(2ndTag) nl nl
             else set(2ndTag)
           endif
           '@OUT INT 2 = ' set(txt) use(1,10)
'\io3 ' > if(2ndTag) nl nl
             else set(2ndTag)
           endif
           '@OUT INT 3 = ' set(txt) use(1,10)

       c Paragraph in an introduction.
'\ip' any(sp,sp)  >  next
'\ip' any(sp)     >  next
'\ip'             >  if(2ndTag) nl nl
                      else set(2ndTag)
                     endif
                     '@PAR INT = ' clear(txt) set(int)
                     use(1,10)
```

Mod 15 Reading from the Keyboard and Writing to the Screen

COMMANDS/TOPICS COVERED

write **read** **wrstore**

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of `read`, `write`, and `wrstore` by modifying and/or writing tables properly using them;
- modify a table to include the display of book and chapter as the input is processed.

INSTRUCTION

1. **Providing user information on the screen—** The `write` command is used to send the character string following it to the screen:

```
"search argument"   >     write "string"
```

The character string may only include `nl`'s and characters in delimiters. The string is terminated by any subsequent *command* (except for `nl` or `c`). The `write` command does not affect the normal direction of output (i.e., the output file or storage area).

The `write` command can be used to provide the user with certain information, as in the following example:

EXAMPLE (from LONGWD.CCT)

```
C ********************************************************
C LONGWD.CCT modified for PAD-CCP, May 1991, by K. Seitz
C -- a table to make list of all words of certain length or
C    longer. For use in setting up a hyphenation table.
C ********************************************************

begin       > store(1) " " nl ",.:;><!\|()$#/'0123456789"
                 endstore
  write nl '*********************************************'
  write nl '*                                           *'
  write nl '*  This table selects all words in a file of  *'
  write nl '*  a certain length or longer. It is helpful  *'
  write nl '*  in deciding which words to hyphenate. Best *'
  write nl '*  when used on a word-list.                  *'
  write nl '*                                           *'
  write nl '*********************************************'
```

2. **Constructing an interactive table with `write` and `read`—** The `write` command can also be used to ask the user for a keyboard response which will then be input to the processing by a `read` command. If storing is in progress, the keyboard response (up to but not including the <ENTER>) will be sent to the storage area; otherwise, it is sent directly to the output file.

EXAMPLE 1 (from LONGWD.CCT)

```
C ********************************************************
C LONGWD.CCT modified for PAD-CCP, May 1991, by K. Seitz
C -- a table to make list of all words of certain length or
C    longer. For use in setting up a hyphenation table.
C ********************************************************

begin       > store(1) " " nl ",.:;><!\|()$#/'0123456789"
                 endstore
  write nl '*********************************************'
  write nl '*                                           *'
  write nl '*  This table selects all words in a file of  *'
  write nl '*  a certain length or longer. It is helpful  *'
  write nl '*  in deciding which words to hyphenate. Best *'
  write nl '*  when used on a word-list.                 *'
  write nl '*                                           *'
  write nl '*********************************************'
  write nl nl 'ENTER LENGTH OF WORDS '
  write       '(IN NUMBER OF CHARACTERS) DESIRED: '
       store(2) read endstore
       store(3) '0' endstore
       store(5) 'abcdefghijklmnopqrstuvwxyz'
                'ABCDEFGHIJKLMNOPQRSTUVWXYZ' endstore

group(1)
     any(5)  > store(4) dup incr(3)
               use(10)
     any(1)  > ''
     ''      > '' fwd(1)
     endfile > endfile

group(10)
     ''      > dup fwd(1) incr(3)
               ifeq(3) cont(2)
                   use(20)
               endif
     any(1)  > store(3) '0' endstore
               use(1)

group(20)
     ''      > dup fwd(1)
     any(1)  > endstore out(4) nl
               store(3) '0' use(1)
```

EXAMPLE 2 (from WRDLST.CCT)

```
c ccccccccccccccccccccccccccccccccccccccccccccccccccc
c CREATE an id line by asking for
c a response from the keyboard.

group(1)
''   >   "\id "
         write nl
         "What goes into the id line?" nl
         read   c This expects a response from the user
         use(2)
```

In the first example, the keyboard response was read into storage area '2' to be used in future decisions/changes in the table. In the second example the keyboard input was not needed for the table processing and went directly to the output file.

(The `read` command causes the processing to halt until the <ENTER> key is pressed.)

3. **Writing storage area contents to the screen with** `wrstore(name)`— Another helpful command for outputting information to the screen is `wrstore(name)`. It causes the contents of the named storage area to be written to the screen.

EXAMPLE (from 7CHAR.CCT)

```
         c ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
         c OUTPUT THE FINAL REPORT
         c   TO THE FILE AND THE SCREEN.
define(Rep) >
    nl out(TotWds) " total words"
    nl out(DelWds) " deleted words (less than 7 chars)"
    nl out(RetWds) " retained words (7 + chars)"
    endfile
    write  nl wrstore(TotWds) write " total words" nl
    wrstore(DelWds)
    write " deleted words (less than 7 chars)" nl
    wrstore(RetWds) write " retained words (7 + chars)" nl
```

The first half of the above `define` outputs the final report to the output file. Then the `write` and `wrstore` commands put the same information to the screen for immediate access. Notice that `wrstore` is a command in itself. If it follows a `write` command, that `write` command is terminated and must be re-issued to output additional information to the screen.

4. **Using** `write/wrstore` **to display on screen the extent of progress—** When a complete New Testament or Bible is input to CC, the processing can take awhile. It is sometimes reassuring to display on

screen the book and chapter number currently being processed. This at least lets you know that the processing is progressing and is not hung in a loop (such as the one cautioned about in Mod 8 involving the null search argument). The example below shows one way that the book and chapter can be written to the screen.

EXAMPLE

```
begin > store(bk) ""              c for book name
        store(bkmsg) nl "Book: "  c book name message
        store(ch) "0"             c chapter no. storage
        store(chmsg) d27 "[40D" d27 "[11C" "Chap: "
                                   c chapter message
        store(chtab) "    "  c overwrite spurious chars
        endstore
...
group(2)
"\id "   > store(bk) fwd(3) endstore   c store book name
           wrstore(bkmsg) wrstore(bk)  c write to screen
"\c "    > store(ch) use(5)
...
group(5)
any(num) > dup
any(sp)  > endstore
           wrstore(chmsg)
           wrstore(ch)
           wrstore(chtab)
...
```

This example only shows those entries necessary for the screen display. These, of course, would be combined with the entries for the major purpose of the table. There are two things worth pointing out in this example:

a. The string stored in ‘chmsg’ in the `begin` is a video screen ‘escape’ sequence that causes the chapter numbers to overlay each other when they are written to the screen rather than appearing side by side. The codes used are specific to certain video screens. Different video screens may require different codes. (Remember, `"d27"` in CC, as seen in this example, means ‘decimal 27’, which is the escape character.)

b. The fixed contents of storage areas ‘bkmsg’, ‘chmsg’, and ‘chtab’ could have been performed as strings following `write` commands within `group(2)` and `(5)`. One reason for putting them into storage areas following the `begin` statement is to make them more accessible in case they need to be modified.

PRACTICE ACTIVITIES and QUESTIONS

1. Write a table that would ask the user for his name, and then respond to the screen with:

```
Hi, (name)! You have successfully completed
this activity. Congratulations!!
```

_____

_____

_____

_____

_____

_____

Since CC requires an input file, write this table so the message will print regardless what the data says (try using a null match). Also be sure to provide for a way to end the program after printing the message once.

2. Assume that a storage area ‘count’ contains the number of verses found in a file. Write a command that would output the message "There were _____ `verses found`" to the screen with the appropriate number being placed in the blank.

_____

_____

READING ASSIGNMENT

CC User’s Guide: 23 read, **26** write, **26** wrstore

EXERCISES

1. Write a table that will:
   a. ask the user for the SFM used for chapter numbers and store it;
   b. ask for the SFM used for section heads and store it;
   c. read the input file EMLK.SFM, counting the number of chapters and section heads; and
   d. output a message both to the screen and in the output file giving the proper counts of chapter and section head SFMs. (You may choose to omit all other text from output or pass it unchanged to the output.)

   `LIST` the input file to determine how to answer the questions for

SFMs used. After successfully running CC, print out your table and hand it in.

2. Modify REFIND.CCT (see Fig. 8. Mod 4; also on your disk) to display the book name and chapter number on the screen as the input file is processed. (You may pattern your modifications from the example shown in this module if you wish. You may try overlaying the chapter numbers or display each chapter number on a new line.)

   After successfully running CC using EMLK.SFM as your input file, print out your table and hand it in.

3. Run CC using the table you created in question 1 of the Practice Activities section above. Use *any* input file.

Mod 16 Calculating with CC

COMMANDS/TOPICS COVERED

add	**mul**	**mod**
sub	**div**	

OBJECTIVES

At the end of this module, the student will be able to:

- demonstrate an understanding of `add`, `sub`, `mul`, `div`, and `mod` by analyzing table logic and providing the results of table entries.

INSTRUCTION

Commands are available for adding, subtracting, multiplying, dividing, and finding the remainder after a division. These all involve storage areas and are replacement side actions. They are not frequently used in publishing preparation, but are quite useful when mathematical calculations are necessary.

1. `add(store1) "number"`— To the number stored in ‘store1’ add the number represented by the string ‘number’, placing the result in storage area ‘store1’.

2. `sub(store1) "number"`— From the number stored in ‘store1’ subtract the number represented by the string ‘number’, placing the result in storage area ‘store1’.

3. `mul(store1) "number"`— Multiply the number stored in ‘store1’ by the number represented by the string ‘number’, placing the result in storage area ‘store1’.

4. `div(store1) "number"`— Divide the number stored in ‘store1’ by the number represented by the string ‘number’, placing the whole-number portion of the result (quotient) in storage area ‘store1’, and discarding the remainder.

5. `mod(store1) "number"`— This is the same as a `div` operation except that the *remainder* portion of the quotient is placed in ‘store1’, and the whole-number portion is discarded.

   The `cont(name)` command could be used in place of `"number"` with any of the math commands.

EXAMPLE

```
...
endfile   >  store(total) outs(SFM_errs) endstore
             add(total) cont(ortho_errs)
             add(total) cont(quote_errs)
             "Total errors: " out(total) nl
             "Orthographic errors: " out(ortho_errs) nl
             "Quote system errors: " out(quote_errs) nl
             "SFM errors: " out(SFM_errs) nl
             store(misused) outs(SFM_errs) endstore
             sub(misused) cont(unident)
             "Unidentified SFMs: " out(unident) nl
             "Misused SFMs: " out(misused) nl
             store(%calc) outs(SFM_errs) endstore
             mul(%calc) "100"
             div(%calc) cont(total)
             "SFM errors (as percent of total): "
             out(%calc) "%" nl
             endfile
```

PRACTICE ACTIVITIES and QUESTIONS

1. Write the output from the above example if SFM_errs = 10, ortho_errs = 7, quote_errs = 3, and unident = 4.

_____

_____

_____

_____

_____

_____

_____

2. Write the additional entries that would be needed to calculate and output a message giving the percentage of SFM_errs caused by unidentified SFMs.

_____

_____

_____

_____

_____

READING ASSIGNMENT

CC User's Guide: 11 add, **14** div, **20** mod, **21** mul, **25** sub, **44-45** Arith. Cmds.