learn

User's Manual

Table of Contents

1. Introduction 1

Introduction to **learn** 1

How to use this manual 1

Documentation conventions 1

2. Using the **learn** utility 3

Introduction 3

Entering the **learn** program 3

Entering the name of the subject 4

Reading the introduction to the subject 4

Following the instructions within the script 4

Changing **learn** subjects 5

3. Writing **learn** scripts 7

Introduction to **learn** script writing 7

Planning the script path 8

Writing the lesson text 14

Constructing the lesson 15

Creating a file 16

Printing instructions 17

Allowing the student to answer 18

Evaluating the student's actual work 19

Evaluating the results of the student's work 21

Recording the student's performance 21

Giving the student another lesson 22

The **learn** directory structure 23

Creating the directory for the new subject 23

Starting the subject with lesson L0 23

The **learn** program commands 24
create file 25
#print 25
#print *file* 25
Terminating a command 26
#end 26
Giving the student control 26
#user 26
Capturing the student's work 26
#copyin 26
#uncopyin 26
Capturing the student's results 27
#copyout 27
#uncopyout 27
Processing the lesson in the shell 27
#pipe 27
#unpipe 28
Testing the student 28
#cmp *file1 file2* 28
#match *pattern* 28
#bad *pattern* 28
#succeed 29
#fail 29
#negate 29
Recording the results 29
#log *file* 29
Giving the student another lesson 29

#next 29
4. Debugging **learn** scripts 31
Examining the lesson file 31
Verifying the command syntax 31
Verifying that necessary files are available 31
Verifying that the next lesson exists 32
Index 33

1. Introduction

Introduction to learn

The **learn** utility is a computer-aided instruction program for the COHERENT operating system. The purpose of **learn** is to train prospective users how to interact with the COHERENT operating system.

By following the **learn** lessons, the student gains "hands on" experience using the COHERENT operating system. **learn** provides step-by-step instructions for the student to follow, and it performs periodic evaluations of the student's performance.

Because **learn** is a COHERENT operating system utility, any COHERENT command can be used in conjunction with the **learn** commands. This structure enables the student to try out COHERENT commands within the **learn** framework.

How to use this manual

This manual explains how to select and use lessons on the **learn** utility as well as how to create new **learn** lessons.

Section 2 is for users who plan to learn COHERENT through the **learn** utility. This section includes procedures for logging in, accessing the **learn** utility, and reading **learn** scripts.

Section 3 steps the novice script writer through the planning and writing of a script, including how to evaluate the progress of student users. It also provides a synopsis of special **learn** commands.

Section 4 provides the script writer with some basic debugging techniques.

Documentation conventions

The examples in this manual are in a "query and response" style: they contain both specific terms that the user must enter into the computer, and the **learn** program responses to user input. When the manual lists these specific terms and phrases, referred to as "literals", they will be presented in **boldface**.

Control characters are letters that are used in conjunction with a special command key. For ease of identification, they are always shown enclosed in angle brackets, e.g. **<ctrl-D>**. When you are

instructed to enter a control character, hold down the CTRL key on your keyboard, while simultaneously typing the indicated letter.

You should not substitute uppercase letters for lowercase letters, and vice versa, when entering information into the **learn** utility because COHERENT distinguishes between uppercase and lowercase. If you do mix up the cases, you will be giving the computer the wrong command.

2. Using the learn utility

Introduction

To use the **learn** utility, you need to have access to a terminal connected to a computer that has COHERENT installed. Ask your system administrator for a log in name and a password. Your administrator can help you to log into the system the first time; for future reference, use the instructions in this section.

Entering the learn program

To enter the **learn** program, type **learn**. Do not forget to press the carriage return key called <**RETURN**>. You will need to do this at the end of each line you type. The **learn** program begins.

After you enter **learn**, the computer gives you some introductory information about **learn**. Figure 2-1 shows the **learn** introduction introduction you will see.

```
Learn provides computer-aided instruction
about various facilities that are available
in COHERENT, allowing you to answer
questions after trying out these commands
in a situation that simulates the normal
COHERENT user interface.
Currently, sample scripts named "editor"
and "filesample" demonstrate some of the
features of learn.
As new scripts are available, you may use
them by typing their subject name.

Enter subject ?
```

Figure 2-1: Introduction to **learn**.

Entering the name of the subject

As you can see in Figure 2-1, the system requests you to enter the name of the subject you want to study.

Currently, there are sample scripts under the subjects of *editor* and *filesample*. To try one of these scripts, type in either **editor** or **filesample**. As other scripts are implemented, the introduction to **learn** will reflect the new study courses.

Reading the introduction to the subject

After you have entered the name of the subject you wish to study, the system presents an introduction to the subject at your terminal. Read the introduction to ensure that the script describes the information you want to study. Then carefully follow the instructions listed.

If you do not wish to continue with the subject, enter **bye** when the '$' prompt appears. The computer responds "Good bye" to acknowledge your request to leave **learn**.

You can type **bye** anytime the computer is waiting for a response. Typing **bye** returns you to the **shell** of the operating system, where you can reenter **learn** to study another subject or log off COHERENT.

Following the instructions within the script

As you progress through **learn** scripts, follow instructions closely to gain the maximum benefit from the lessons provided. Each lesson consists of instructions, experimentation, and evaluation. This combination of COHERENT system resources increases your proficiency as you progress through the **learn** program.

When you give one or more incorrect answers while studying a lesson, **learn** gives you a choice of repeating the lesson or trying an alternate lesson. If you fail a lesson, do not be discouraged; try the lesson again, or type **bye**, log off, and try again later.

At the end of each accurately completed lesson, the **learn** program automatically presents an appropriate follow-up lesson until you have mastered a subject. If you wish not to continue in the subject at this time, enter **bye** when the '$' prompt next appears.

Changing learn subjects

To select another subject, simply complete the following three steps:

```
1. Type "bye" to exit the current subject.
2. Type "learn" to present the topics for study.
3. Type the name of the new subject.
```

You now can follow the lessons in the new subject.

3. Writing learn scripts

Introduction to learn script writing

In this section, you will learn the basic skills you need to write scripts for the **learn** utility. Throughout this section, examples using segments of a sample **learn** course illustrate a way that courses may be presented to the **learn** student.

The purpose of **learn** is to direct the student to understand and use COHERENT commands. The student begins with simple scripts and progresses to expert proficiency, simply by following the logical structure of the course. At critical points during the course, the student's record is measured to direct the student to easier or more advanced lessons.

Many sequences or tracks of lessons may be provided to allow for the speed and ability of various students. A three-track course could be structured as follows:

Track a	*Track b*	*Track c*
2.1	2.1	2.1
2.2		
2.3	2.3	
2.4		
2.5	2.5	2.5
2.6	2.6	
2.7		
2.8	2.8	
2.9		
3.0	3.0	3.0

Table 3-1: A three-track course.

The student's performance during Lessons 1.0 through 1.9 determines the track which the student begins to follow from Lesson 2.1 to Lesson 3.0.

The student is not locked into a particular track. A student who starts out poorly could perform the new material well and be shifted to a faster track. Conversely, someone who progresses

easily through preliminary lessons but has trouble with more difficult material can be directed to a slower track.

Planning the script path

As the **learn** script writer, you must decide which skills are prerequisite to other skills and then develop the curriculum to take advantage of the student's previous experience.

It may be a good idea to chart out the material for the entire subject course before planning even the first lesson. This preparation enables you to determine if a course is too long to be completed during a normal **learn** session, if too much information will be covered, or if the topics of the course are too diverse.

In setting up a course, you can use three different types of questions to teach your students:

- Yes or no
- Simple answer
- Open-ended

Yes-or-no and simple-answer questions should dominate the slower tracks of a subject; use the open-ended question to give students problems that use the knowledge they gained in the easier lessons. Based on *Track a* in figure 3-1, the example below shows how the three question types can be used.

Question	*Lesson*
Concept	2.1
yes or no	2.2
yes or no	2.3
simple answer	2.4
open-ended	2.5

The *yes-or-no question* is the simplest type to answer. Using this type of question early in the course emphasizes for the student how easy it is to use the **learn** program.

In Figure 3-1, you see a sample yes-or-no question from the sample **learn** course, **filesample**.

```
#print
Have you read the instructions since beginning
the learn program?  Type "yes" or "no".
#user
#match yes
#fail
To make the most of your learn training,
you must read all instructions.  Here are
the first instructions again:
#log/usr/learn/log/who-reads
#next
0a 0
1.02 1
```

Figure 3-1: Lesson 1.01.

The elements of Lesson 1.01 are:

- A print command
- Text to be printed
- A command to give the user a chance to respond
- A command to match the user's response
- A conditional print command
- Text to be printed upon failure of the lesson
- A command to record the results of the lesson in a specified file
- A command to direct the program to the appropriate follow-up lesson
- Lesson numbers for possible follow-up lessons

All of the commands in this lesson are **learn** commands. The **learn** commands begin with the '**#**'. The **#print** command prints the subsequent lines up to the next **learn** command at the student's terminal. The **#print** command in this lesson gives the student instructions to complete the lesson.

The **#user** command turns control over to the student. At this point, the student may use any COHERENT command. The **learn**

program regains control when the student enters **yes, no, answer,** or **ready**.

The **#match** command checks short answers against the correct answer to evaluate the performance of the student. In this case, the correct answer is yes.

The **#fail** command is a conditional print command. The subsequent lines (up to the next **learn** command) are printed only when the student enters an incorrect answer.

The **#log** command tells the program to record the results of the lesson in a log file. In this case, the script writer is keeping track of how many students read the instructions during their first **learn** session. The results are stored in a special file called *who-reads*.

The **next** command directs the program to the next appropriate lesson for the student to try, based on performance of the previous lesson. The two lines after **#next** indicate the possible options. If the student's score is closer to 0, Lesson 0a will be given next. When the student's score is closer to 1 than 0, Lesson 1.02 is selected.

With the *simple-answer question,* you are moving the student one rung up the difficulty ladder. By answering this type of question, the student exhibits skills that you can evaluate more subjectively. In a way, you are even grading the typing ability of the student.

Figure 3-2 shows a sample simple-answer question from the **learn** filesample course.

```
#print
What key or keys must you press to
transmit a response to the computer?
Type "answer [name]" where [name] is
the key or group of keys you enter.
#user
#match [R,r]eturn
#match RETURN
#fail
Here is a hint:
You do not type this answer; you would
press the key with this name.
Try again.
If you get the answer wrong next time,
type "bye" and reread your manual.
#succeed
You have completed the prerequisites.
#log /usr/learn/log/return
#next
1.02 0
1.02a 1
1.1 2
```

Figure 3-2: Lesson 1.02.

Like Lesson 1.01, this lesson uses the **#print** command and prints the subsequent lines, then gives the user control to answer the lesson. In addition, the student has three possible correct answers to match and receives a printed message upon successful completion of the lesson.

Within the text after the **#print** command, the script writer tells the students to preface their answers with the word **answer**. Typing **answer** is necessary because the answer is more than **yes** or **no**. **answer** tells the **learn** program that it now controls the script again.

The **#match** command is used twice in Lesson 1.02 to include a greater number of correct answers. The first **#match** command passes the student to the next lesson if either **return** or **Return** is entered. The second **#match** command passes the students to the

next lesson when they enter **RETURN**. Therefore, three answers are acceptable to the script writer for this lesson.

The **#succeed** command is a conditional print command that causes the subsequent lines to be printed when the student enters a correct answer.

Notice that **learn** directs the student to one of three possible next lessons after evaluation of the lesson. The current lesson is repeated if the student has an accrued score of 0; this lesson will continually be presented until the student enters a correct answer.

Because the student can score one point for each lesson of the two lessons presented, the student can have a perfect score of 2. When the score is 2, Lesson 1.1 is next presented.

If the student had difficulty with 1.02, Lesson 1.02a is next.

The *open-ended question* is the most difficult of the questions for students. It also provides results that give you the most subjective evaluation of the student.

The open-ended question requires that the student assess the exercise and make a complete answer. Students may make several attempts at the answer before entering their final response.

Figure 3-3 shows an elementary open-ended question.

```
#create .eval
This is my first file on COHERENT.
#print

In this exercise, you will use "cat" to print
the file you created in Lesson 1.1.

To use "cat" to print a file, type "cat
filename", where "filename" is the name of the
file you want printed.  If you are not sure of
the name of the file, you may type "lc" to see
a list of the existing files before you select
the file for printing.

When you are finished, type "ready" to let me
evaluate your work.
#copyout
#user
#uncopyout
tail -1 .ocopy >rest
#cmp rest .eval
#log
#next
4 2
1.3 4
```

Figure 3-3: Lesson 1.1.

Lesson 1.1 introduces four new **learn** commands: **#create** *file*, **#copyout** , **#uncopyout** , and **#cmp** [fileP] [file2]. In addition, one COHERENT command is used.

The **#create .eval** statement creates the file **.eval** in the temporary play subdirectory of the **learn** directory. This file will contain the subsequent lines of text in the lesson. The file **.eval** is the correct answer to this exercise.

When circumstances require that the student type in an answer that is longer than one line, he must follow a certain protocol. By invoking the **#user** command, control passes temporarily from the

script to the student. When the student is finished keying in his response, he must cue the **learn** program that program control is returned to the script. He does this by typing **ready**.

The **#copyout** and **#uncopyout** commands must be used together—like book-ends—to obtain the desired results from your lessons. When these commands are used on either side of the **#user** command, anything sent by the computer to the student's terminal is copied to the file **.ocopy**. When the student finishes the exercise, you may manipulate the contents of **.ocopy** to evaluate the student's work.

The command

```
tail -1 .ocopy >rest
```

finds the last line of **.ocopy** and copies it into the file named rest. Since the student can type any COHERENT command, many different lines may appear in **.ocopy**. For example, some students may forget the name of the file they created in the previous lesson and use the **lc** command to refresh their memories. If this happens, a line of file names will appear in **.ocopy**. Therefore, script writers use this command to copy the last line of the computer output to a separate file for evaluation.

The command

```
#cmp rest .eval
```

completes a line-by-line comparison of the two files named, to verify that the student's work is correct. The results of this command determine whether the student passes or fails the lesson.

Writing the lesson text

To best use the "repetition and reinforcement" method of instruction, write scripts using language that is easy to understand. Use simple wording. Achieving this goal may mean that, in an early lesson, you only partially define a concept.

For example, in Figure 3-4, the lesson tells the student that there are several uses of command **cat** but explains only one.

```
One of the most frequently used file commands
is "cat". The "cat" command can be used
to create, print, and combine files. In this
exercise, you will create a file using the
"cat" command. When the "$" prompt appears,
type the following lines:

     cat >firstfile
     This is my first file on COHERENT.
     <CTRL-D>

When you are finished, type "ready" to let me
evaluate your work.
$
```

Figure 3-4: Lesson 1.1 as the student sees it.

Constructing the lesson

The structure of the **learn** program gives you the ability to test the student on using the facilities of the COHERENT operating system, grade the student's progress, and plan future lessons.

Here is the basic structure of a script, which the **learn** program reads and processes a line at a time:

Command	*Description*
#create *file*	A command to create a file whose contents are the correct answer(s) to the lesson.
#print *file*	A command to print instructions to the student.
#user	A command to give the user control to enter an answer for the lesson. (The script regains control when the user enters yes, no, answer, or ready.)

#match *pattern*	A command to test the student's work.
#bad *pattern*	A command to test the student's work.
#cmp *file1 file2*	A command to test the student's work.
#log	A command to post the results in the log file.
#next	A command to give a new lesson if the student passes the present one, and to repeat the present lesson if the student fails.

The **learn** program interprets the **learn** script for the student. **learn** reads each line of the script and, where necessary, prints it for the student to act upon.

The script is made of a combination of the nineteen **learn** commands, as well as COHERENT commands that the script writer selects.

Because **learn** is a COHERENT utility, any COHERENT command is valid in a **learn** script. Specialized **learn** commands that perform COHERENT operations are available to lower overhead within the program. Later in this section is an explanation of each of the specialized **learn** commands.

Each **learn** lesson is constructed of the elements listed above. In the following sections, you will learn how to construct the lesson using those elements.

Creating a file

Creating a file is useful in two ways: storing correct responses for comparison with student responses; and saving instructions or other information that you want to print repeatedly in one or several scripts.

When the lesson you are writing requires that you create one of these files, you can use the **#create** file command. This command is executed by the program while the program is printing the instructions to the student.

learn stores in the new file all the text in the script following the **#create** command, until this process is completed by invoking the

next command that begins with '#'. To use the **#create** file command, follow the format described below.

```
cat L1.1
#create .test
This is a test.  This is only a test.
If this were a real emergency, this would not
be a test.
#user
<CTRL-D>
```

Figure 3-5: Example of the **#create** file command with a sample file.

To create and execute this example, use the **cd** command to access the **pieces** directory. Type:

```
cd /usr/learn/lib/pieces
```

Use the **cat** command to create the file L1.1 as shown in Figure 3-5. When you have finished typing the example, you can execute it by entering the **learn** subject, **pieces**. Because the lesson does not print text, you may not know whether the '$' prompt is the shell prompt in the play area of **learn** or the shell prompt of the **pieces** directory. Type **lc** to see what files are in the current directory. If the file **.test** (which was created by the lesson you just wrote) is in the directory, print it using **cat**. If **.test** is not present, you are not in the user area of **learn** and should type **learn** and **pieces** again to try the lesson.

After you have created any files you need for evaluating the student's response, you can enter the instructions that **learn** is to print for the student.

Printing instructions

As shown in Figure 3-6, the instructions are printed using the **#print** command. The **#print** command, like the **#create** *file* command, occupies a line by itself. When a file name is specified, the file is printed. When no file is specified, any text that follows the **#print**

command is printed or displayed at the student's terminal, up to the next command beginning with **#**.

Figure 3-6 is created by typing the following lines:

Command	*Description*
ed L1.1	Puts file L1.1 into the editor.
a	Appends (writes new lines) to the end of the file.
#print .test	New test line.
.	Ends append function.
w	Writes new file contents to L1.1.
q	Quits the editor.

To execute the lesson as it exists now, enter the **learn** subject, **pieces**. When the '$' prompt appears, type **ready** to give control to the **learn** program; then watch what happens.

```
#create .test
This is a test.
This is only a test.
If this were a real emergency, this would not
be a test.
#user
#print .test
```

Figure 3-6: Example of **#print** command.

Your next step is to prepare **learn** for receiving the student's response and to capture that response for evaluation.

Allowing the student to answer

To instruct **learn** to expect a response, use the **#user** command. This command occupies a line by itself and requires no argument.

After executing the **#user** command, the **learn** program reads the student's response. If the student enters one of these four responses: yes, no, *answer*, or ready, **learn** continues processing the script.

You can now expand the sample lesson to include a respond section for the student. Figure 3-7 is a further expansion of the example lesson. To create this lesson, use the remove command **rm** to eliminate the current lesson and then recreate it, or you can enter the editor and add, delete, and change lines to update the lesson.

```
#create .test
This is a test.  This is only a test.
If this were a real emergency, this would not
be a test.
#print
Type "cat .test" and see what happens.  Type
"ready" when the "$" prompt reappears.
#user
#print
Did you see a two-line message? Answer "yes" or "no".
#user
#match yes
#log
```

Figure 3-7: Example of **#match** method of evaluating the student's yes-or-no response.

When the desired response of the student is other than yes or no, you need a way to capture the response for comparison with the correct answer.

Evaluating the student's actual work

In evaluating the student, you can choose between evaluating the actual work of the student or the results of the student's work.

Several commands are available to help you evaluate the student's actual work. To evaluate short answers, you can use the **#match** *pattern* command. The **#match** *pattern* command compares the last line of the student's input with the designated pattern. (As with many commands, there is a default case; in this instance, if you do not specify a pattern, any response will match.)

You also can set up a filter to capture frequently entered wrong answers by using **#bad** *pattern*. The purpose of this filter is to enable the computer to give the student hints for entering answers

that the computer can accept. For example, the answer "maybe" could cue the computer to come up with a helpful prompt, such as: "just answer with a yes or a no". When the student's answer matches the bad pattern, you can enter your hints as prompts on the following lines, up to the next **learn** command.

For longer answers (like those in response to open-ended questions), you must first capture the actual work of the student by using the **#copyin** command before the **#user** command and the **#uncopyin** command after **#user**. This bracket of commands traps everything the user types at the terminal and copies it at the file **.copy**. After the student finishes the example or answers the question, the script can compare the student's work with a file you created at the beginning of the script.

To make the sample lesson a lesson that compares the student's response to a predetermined correct answer, enter the editor and modify the file to include the contents shown in Figure 3-8. Notice the new file you are creating, and the files being compared.

```
#create .eval
cat .test
ready
#create .test
This is a test.  This is only a test.
If this were a real emergency, this would not
be a test.
#print
Type "cat .test" and see what happens.  Type
"ready" when the "$" prompt appears.
#copyin
#user
#uncopyin
#cmp .copy .eval
#log
```

Figure 3-8: The use of **#copyin** and **#cmp** in evaluating the student's work.

To compare the student's work in .copy with the file you created at the beginning of the script, use the **#cmp** *file1 file2* command. In the example above, the line is:

```
#cmp .copy .eval
```

Remove file **L1.1** when you are finished testing your scripts by typing:

```
rm /usr/learn/lib/pieces/L1.1
```

Evaluating the results of the student's work

To capture the results of the student's work, use the bracket of commands **#copyout** and **#uncopyout** around **#user**. This set of commands traps anything that the computer prints at the student's terminal and copies it into the file *.ocopy* for comparison with the correct answer. This lists not only the commands used by the student, but also the output of each command (e.g. if he asked for a list of his files, that list is also captured.) Results-oriented educators prefer this method of evaluation.

Recording the student's performance

Now that the student's work has been captured and compared with your correct answer, you can log the results in the log file. The log file is a file in the directory **/usr/learn/log**. It has the same name as the subject for which the logging is done. Ordinarily, this file is created by the **learn** program the first time the **#log** command is used in a subject. The contents of the log file are:

```
- The number of the lesson
- The name of the user
- The word "right" or "wrong" describing the
  student's response
- The accumulated score, called speed=n
- The day, date, and time of lesson completion
```

Each lesson's record is appended to the log file specified by the script writer. This enables the script writer to examine the results of the lessons over as long a period as is necessary to evaluate the effectiveness of the course.

You can record the completion records of special lessons in a separate log file. This file can contain the log for one or several lessons, and can be located in any directory.

To create the log file, use the **#log** command to have the log information put into the subject log file, and use the **#log file** command to send the information to a special file; **file** must be the absolute file name, as in **/usr/learn/log/test**.

To log the results of the lesson in a file that is not the default subject log file, enter **#log file** on a line following the line on which the results were tested. For example:

```
#log /usr/learn/log/spotcheck
```

logs the results of the lesson in the file named **spotcheck** instead of the log file with the subject's name.

Giving the student another lesson

You may now give the student a new lesson based on sufficient ability to complete the previous lesson. By using the **#next** command, you can test the results of the lesson against the prerequisites for the next lesson. Based on these results, **learn** then presents the student with the next lesson.

After testing the student, you can establish multiple paths for the student with the **#next** command. The student's score is incremented by 1 for each correct answer and decremented by 4 for each incorrect answer. The score is always between 0 and 10, inclusive.

Figure 3-9 shows the uses of the **#next** command. The first use tests for accuracy early in the session, and the second tests for the accumulated score later in the session.

```
#next                 #next
1.1 0                 4.2a 10
1.2 1                 4.2b  6
                      4.2c  2

  A                     B
```

Figure 3-9:
(A) Early next lesson options.
(B) Later next lesson options.

The learn directory structure

Now that you have formulated the lessons you want **learn** students to try, you may enter the lessons into the COHERENT operating system.

Creating the directory for the new subject

Before you can start executing **learn** scripts, you must first make a directory to contain your lesson files. This is the only directory you must make to have an operating **learn** course. The log directory or the play directory are created when a **#log** or **#user** command, respectively, is interpreted by the **learn** program.

To make this directory, first use the **cd** command to change to directory **/usr/learn/lib**. Once you are in the **lib** directory, you can make the directory for your subject. The name you use here will be the subject that the student types when asked, **Enter subject?**.

To make a directory for the subject cat, enter the following:

```
mkdir cat
```

The next time you list the contents of the **lib** directory, there will also be a subdirectory named **cat**.

Starting the subject with lesson L0

The **learn** program looks for Lesson L0 to start each subject. Therefore, you must create a file called **L0** in each subject's direc-

tory. **L0** may contain a complete introduction to the subject, or merely a statement to direct the program to the first lesson.

Figure 3-10 shows an example of a full introduction in file L0.

```
#print
Welcome to the exercises on files.  This
segment of your training concentrates on
creating, printing, modifying, and removing
files from COHERENT.

Before exploring files, we will test you for
prerequisite skills.

If you wish to leave learn to come back later,
type "bye" after the "$" prompt appears.

#next
1.01
```

Figure 3-10: Lesson L0 of the filesample subject.

The L0 file of the editor subject is very brief. The text of this script is shown in Figure 3-11.

```
#next
1.1
```

Figure 3-11: Lesson L0 of the editor subject.

The learn program commands

This section describes the 19 commands that are part of the **learn** utility. Each **learn** command begins with the **#** character. Every line that follows a **learn** command is interpreted as being part of that command. Another **learn** command or a COHERENT command causes one operation to stop and another to begin.

create file

This command creates **file**, and writes the lines between this command and the next **learn** command in **file**. For example:

```
#create text
Text
for
file
#print
```

Use this command to create and initialize working files and reference data for the lesson.

#print

This command prints or displays the lines between this command and the next **learn** command at the student's terminal. Use this command to print instructions and explanations for the student. For example:

```
#print
Now is
the time
for all....
#user
```

#print *file*

This command prints the contents of **file** at the student's terminal. This command is useful when the text the student is to see is repeated within the script or from one script to another. Put the text to be repeated into **file** using the **#create file** command. Then print the contents of **file** whenever necessary.

Terminating a command

#end

This is a null command that serves as a terminator for another **learn** command when nothing else is appropriate.

Giving the student control

#user

This command gives control to the student. Each line that the student types is executed as any COHERENT command would be. The **learn** program regains control when the student enters **yes, no, answer [answer]**, or **ready**. The student may also terminate the lesson at this time by entering **bye**.

Capturing the student's work

#copyin

This command opens a file called **.copy** to receive anything the student types, holding it for comparison with a control file. To use this command to catch the student's work, **#copyin** must precede the **#user** command.

#uncopyin

This command terminates the **#copyin** command. For the student's work to be captured into the file **.copy**, this command must follow the **#user** command. Example:

```
#copyin
#user
#uncopyin
```

Capturing the student's results

#copyout

This command opens a file called **.ocopy**. This file captures the material that the system prints at the student's terminal for comparison with a control file. To use **#copyout**, it must precede the **#user** command.

#uncopyout

This command terminates the **#copyout** bracket. **#uncopyout** must follow the **#user** command. Example:

```
#copyout
#user
#uncopyout
```

Processing the lesson in the shell

#pipe

The **pipe** command allows you access to the COHERENT shell while remaining in **learn**. A subshell is set up: your script statements are passed to the COHERENT shell in a stream instead of individually. For example, if a **learn** lesson is trying to teach you how the editor works, you can get "hands on" experience. By utilizing the **#pipe**, you are moved temporarily to the editor, allowed to manipulate and edit as much text as you want, and then, by stopping the **#pipe** command (with **#unpipe**), brought safely back to the **learn** lesson you started from.

When you want to evaluate a student's work, the **#pipe** command must be inside the **#copyin** or **#copyout** brackets. Figure 3-12 shows a possible series of commands that includes **#pipe**.

```
#copyout
#pipe
ed
#user
q
#unpipe
#uncopyout
```

Figure 3-12: Automatic entry and exit (quit) of editor with results copied into .ocopy.

#unpipe

This command terminates the **#pipe** command, and also must be within the **#copyin** or **#copyout** brackets.

Testing the student

#cmp *file1 file2*

This command compares the contents of **file1** with the contents of **file2**. Use this command to compare **.copy** or **.ocopy** with your control file. You may also use this command with only one argument to compare a student-generated file with the lines after **#cmp**, up to the next **learn** command.

#match *pattern*

This command compares the last line of student input with the specified pattern. The special characters '*', '?', '[...]', and '\' perform the same functions in **learn** as in file name expansion in the shell. After a correct match, any lines that are not **learn** commands are printed for the student.

#bad *pattern*

This command is the opposite of the **#match** command. Use this command to check for specific wrong answers and then print hints to help the student pass the lesson.

#succeed

This command allows a congratulatory prompt to be printed at the student's terminal, upon correct completion of a lesson. The correctness of the complete lesson is determined as the result of invoking one of the following commands: **#match**, **#bad**, or **#cmp**.

#fail

This command allows a negative prompt to be printed at the student's terminal, as a consequence of an incorrectly completed lesson. As above, the lesson is evaluated by one of the following commands: **#match**, **#bad**, or **#cmp**.

#negate

This command reverses the success and failure states of a lesson. The main use is for lessons that direct the student to send a command to the shell. If the student matches the wrong answer, but the student's response is correct, the status must be reversed.

After the student has finished the lesson and returned control to the script, the script writer may use COHERENT commands such as **tail −n** and **grep** to help evaluate the success or failure of the lesson.

Recording the results

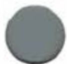
#log *file*

This command writes a log entry about the lesson into the specified file. You must use the absolute file name for this command to work. If no argument is included, the log entry is written in the logging file named for the subject (**/usr/learn/log/subjectname**).

Giving the student another lesson

#next

The lines that follow this command direct the **learn** program to provide follow-up lessons for the student based on the performance of the student on the current lesson.



4. Debugging learn scripts

From time to time while testing your **learn** scripts, you may find that the script does not work as you expected. This section provides guidelines for making the nonoperating script an operating script.

Examining the lesson file

The information you need first is a listing of the commands in the script. Print this listing by using one of the COHERENT commands, or by entering the editor and printing:

```
1,$p
```

Now that you can see the script that **learn** is reading, you can begin to solve the problem.

Verifying the command syntax

Perhaps the easiest check you can perform is to analyze the use of **#copyin** and **#uncopyin**, **#copyout** and **#uncopyout**, and **#pipe** and **#unpipe**.

After verifying that all the open brackets that occur before the **#user** command have terminators after **#user**, you can check to see that any files mentioned in the script actually exist, and are accessible to the **learn** program.

Verifying that necessary files are available

If **learn** cannot find a file specified by a particular script, the student will receive an error message when the script tries to open or locate the file. To see what files exist in the directory **/usr/learn/play/___/**, use the **lc** command when the '$' prompt appears during the problem lesson. If the file that the lesson requires is not in the play directory, make sure that the absolute file name is given for each file **learn** is to print or compare during the lesson, or create the file (this puts it in the play directory) at the beginning of the script.

If **learn** still encounters errors when trying to open a file, you may need to check earlier scripts within the subject to see if the file should have been created there.

Verifying that the next lesson exists

When the lesson you write contains the **#next** command, the **learn** program will return the following error message if the requested lesson is missing:

```
learn: script error--lesson 'N.N' not found
```

A quick fix for this bug is to create a lesson with the name referred to in the **#next** statement. Write the **#end** command into the new file. When you are ready to write the lesson, replace **#end** with that lesson.

Index

#: 16, 18, 24
#bad: 19, 28-29
#cmp: 13-14, 16, 20-21, 28-29
#copyin: 20, 26-28, 31
#copyout: 13-14, 21, 27-28, 31
#create: 13, 15-17, 25
#end: 26, 32
#fail: 10, 29
#log: 10, 16, 21-23, 29
#match: 10-11, 15, 19, 28-29
#negate: 29
#next: 16, 22, 29, 32
#pipe: 27-28, 31
#print: 11, 15, 17-18, 25
#succeed: 12, 28
#uncopyin: 20, 26, 31
#uncopyout: 13-14, 21, 27, 31
#unpipe: 27-28, 31
#user: 13-15, 18, 20-21, 23, 26-27, 31
bye: 4
change subjects: 5
follow-up lesson: 4
learn introduction: 3
leaving **learn**: 4
log in: 3
next: 10
question: 8
    open-ended: 12
    simple answer: 10
    yes or no: 8
repeat a lesson: 4
scripts: 4, 16
    writing: 7

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: ______________________________

Company: ______________________________

Address: ______________________________

____

Phone: ________________ Date: ________________

Version and hardware used: ______________________________

Did you find any errors in the manual? ______________________________

____

____

Can you suggest any improvements to the manual? ______________________________

____

____

Did you find any bugs in the software? ______________________________

____

____

Can you suggest improvements or enhancements to the software?

____

____

____

Additional comments: (Please use other side.)