ESD-TR-75-58

MIXED-INITIATIVE TUTORIAL SYSTEM TO AID USERS OF THE ON-LINE SYSTEM (NLS)

Mario C. Grignetti
Laura Gould
Catherine L. Hausmann
Alan G. Bell
Gregory Harris
Joseph Passafiume
Bolt, Beranek and Newman, Inc.
50 Moulton Street
Cambridge, MA

30 November 1974

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION
L. G. HANSCOM AIR FORCE BASE, BEDFORD, MA 01731

LEGAL NOTICE

When U.S. Gavernment drawings, specificatians ar other data are used far any purpose ather than a definitely related gavernment procurement aperatian, the gavernment thereby incurs na responsibility nar any abligation whatsaever; and the fact that the gavernment may have farmulated, furnished, ar in any way supplied the said drawings, specifications, ar other data is not ta be regarded by implicatian ar atherwise as in any manner licensing the halder ar any other persan or conveying any rights or permission to manufacture, use, or sell any patented inventian that may in any way be related thereto.

OTHER NOTICES

Da nat return this copy. Retain ar destroy.

"This technical report has been reviewed and is approved for publication."

Sylvia R. Mayer

SYLVIA R. MAYER/GS-14
Project Scientist

Sylvia R. Mayer

SYLVIA R. MAYER/GS-14
Task Scientist

FOR THE COMMANDER

FOR P.R. Keckley

ROBERT W. O'KEEFE, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command & Management Systems

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Dete Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-58	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* MIXED-INITIATIVE TUTORIAL SYSTEM TO AID USERS OF THE ON-LINE SYSTEM (NLS)		5. TYPE DF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mario C. Grignetti, Alan G. Bell, Laura Gould, Gregory Harris, Catherine L. Hausmann, Joseph Passafiume		8. CONTRACT OR GRANT NUMBER(s) F19628-74-C-0088
9. PERFDRMING DRGANIZATION NAME AND ADDRESS Bolt, Beranek, and Newman, Inc. 50 Moulton Street Cambridge, MA		10. PRDGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element - 62702F Project No. 2801 Task 04.03
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Mangement Systems Hanscom AFB, MA 01731		12. REPORT DATE 30 November 1974 13. NUMBER OF PAGES 131
14. MDNITDRING AGENCY NAME & ADDRESS*(if different from Controlling Office)* N/A		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15e. DECLASSIFICATIDN/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTIDN STATEMENT *(of the ebstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NDTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Artificial Intelligence, Computer Assisted Instruction, Natural Language Processing, Semantic Grammar, Semantic Network, Tutorial Supervision, On-Line Assistance, Question Answering

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

NLS-SCHOLAR is a prototype system that uses Artificial Intelligence techniques to teach computer-naive people how to use a powerful and complex editor. It represents a new kind of Computer Assisted Instruction (CAI) system that integrates systematic teaching with actual practice, i.e., one which can keep the user under tutorial supervision while allowing him to try out what he learns on the system he is learning about. (over)

DD FORM 1 JAN 73 1473 EDITION DF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Dete Entered)*

20. NLS-SCHOLAR can also be used as an on-line help system outside the tutorial environment, in the course of a user's actual work. This capability of combining on-line assistance with training is an extension of the traditional notion of CAI.

The techniques used in NLS-SCHOLAR are general and can be applied to the teaching of a wide variety of computer related activities.

TABLE OF CONTENTS

PREFACE

The United States Air Force is relying more and more on computer based systems for many of its management, logistics, resource allocation, planning, and command and control functions. Many of these computer based systems are extremely powerful, but part of their large potential usefulness remains untapped because of their complexity.

A case in point is the oN Line System (NLS), a powerful tool for planning and communication developed by the Augmentation Research Center of the Stanford Research Institute. NLS is a computer based system for writing, editing, publishing, and disseminating information of all kinds. Many Governmental agencies including several Air Force facilities, access it through the ARPA computer network. NLS is currently being evaluated by the Air Force as a paradigm for the use of a computer based Management Information System. In particular, one group at the Rome Air Development Center is using NLS experimentally as part of that evaluation.

Although NLS is a complex system providing many options to its users, those who have become proficient with it find it very easy and powerful to use. However, gaining that proficiency is usually very difficult and time consuming, and there is a real need for computer aids to help people with the learning process.

Recognizing the generality of this problem, the Air Force has established as one of its Technical Needs the development of computer based training and decision aids to help people learn how to use these systems. In the following report we describe the work done at Bolt Beranek and Newman on a system, called NLS-SCHOLAR, designed to meet that Technical Need: ESD-TN-Human Performance Aiding in Command, Control and Management Data Systems.

SECTION I - INTRODUCTION

What is NLS-SCHOLAR

NLS-SCHOLAR is a quasi-operational CAI system that uses Artificial Intelligence techniques to help people learn how to use the powerful structural editor of NLS. NLS, the oN Line System [1] developed by Douglas Engelbart and his co-workers at the Augmentation Research Center of the Stanford Research Institute, is a sophisticated modular system which is being increasingly used as an aid in writing, re-organizing, indexing, publishing, and disseminating information of all kinds. It is a very large system made up of many subsystems, and NLS-SCHOLAR deals with EDIT, its most important and most frequently used subsystem.

SCHOLAR, conceived and first developed by the late Jaime R. Carbonell, is an interactive mixed-initiative CAI system that deals with the geography of South America. It is capable of answering freely interspersed questions posed by a student in the course of a tutorial session, and it uses teaching strategies similar to those of good human tutors [2,3,4].

In trying to apply SCHOLAR to other domains of knowledge, such as computer networks [5] and structural editors, we have uncovered new problems that require

radically different approaches. Therefore, NLS-SCHOLAR, although preserving the flavor and interaction characteristics of SCHOLAR, is an almost entirely new system, its underlying philosophy and approach owing much to Brown's SOPHIE system [6,7].

Why an NLS-SCHOLAR system

NLS is a powerful system for preparing and distributing documents that offers many rewards to people who have learned how to use it well. However, its complexity and the multiplicity of its options make learning NLS difficult, time consuming, and at times discouraging for people who are not habitual computer users. With the increased availability to NLS now afforded by the ARPA network and, above all, with NLS playing a fundamental role in the NSW project [8] there is a real need for on line aids to help non-programmers both to learn and to use NLS.

NLS-SCHOLAR is designed so that it may perform either as an on-line helper and question answerer, or as a tutor. When used as a tutor, NLS-SCHOLAR behaves in a very friendly way: in the course of a lesson, students can ask questions, proceed at their own pace, make mistakes safely, ask for help, and give up and be rescued by the system.

In tutorial mode learning is made easy and comfortable by relying heavily on ostensive teaching. New information

is presented to the student by means of an expository part, presenting examples and showing students how to do things. The tutor lets students edit text by themselves and helps them correct their mistakes; it answers questions or performs commands posed by students in a comfortable subset of English; it asks questions and evaluates students' answers; and it presents tasks for students to perform which are then evaluated and commented upon.

As with most CAI systems, learning takes place in complete privacy; students are left alone in a tete-a-tete with the system with nobody witnessing their mistakes, ignorance, or lack of sophistication in the use of computer systems. This relaxed (and relaxing) situation helps the learning process enormously. But unlike most CAI systems, NLS-SCHOLAR is designed so that it can also be used as an on-line help system, so that users can ask questions arising in their actual work with NLS, and expect NLS-SCHOLAR to be aware of what they are doing and answer accordingly. This is especially useful for sporadic users, or for people who have not used NLS for a long time and have forgotten some of its conventions.

NLS-SCHOLAR is designed to be a stand-alone tutorial and help system. A student's prior knowledge requirements are simply to know how to log in, and follow the basic instructions contained in a 2-page handout. The information

contained in these instructions is itself a part of the system's domain of knowledge. For example, the student is told how to erase a character that he has typed, but if he forgets how to do it he can ask NLS-SCHOLAR.

Demonstrating NLS-SCHOLAR's capabilities

The flavor of NLS-SCHOLAR is best conveyed with the help of a demonstration protocol which was actually obtained on-line using the latest version of the system. First, a few helpful comments.

The demonstration of interactive capabilities we want to perform cannot be done "in vacuo"; questions asked by students or by the system, as well as tasks proposed and evaluated arise more naturally and make more sense in the course of a lesson. When used as a tutor, the system is driven by a fixed Agenda which presents to the student carefully sequenced morsels of NLS knowledge and know-how. Since this is a demonstration protocol, our "student"* is very obliging and does the appropriate things at the right times to make explicit specific characteristics of the system.

NLS-SCHOLAR uses two bodies of text as its working examples, one a breakfast menu and the other a dinner menu. In the course of a lesson, students learn how to change the

---

*Actually one of the authors.

contents (and appearance) of these menus by performing editing operations. Menus were chosen as examples because of their direct appeal and general intelligibility, their natural hierarchical structure, and the shortness of their entries which makes them very easy to work with.

In the interest of brevity*, the protocol starts at a point well along in the student's learning of NLS -- he has been told about NLS files, how to load them, print them, delete and insert statements in them, etc. He is about to be taught how to use the Substitute command to effect a change in the breakfast menu (see Figure 1).

Readers familiar with NLS may fail to recognize it as the system depicted in the protocol. This is because NLS-SCHOLAR teaches the use of a newly emerging version of NLS, which is not yet generally available.

---

*For a complete demonstration protocol, see the Appendix.

FIGURE 1 - THE BREAKFAST FILE

```
1  JUICE
   1A  ORANGE
   1B  GRAPEFRUIT
2  CEREAL
   2A  OATMEAL
       2A1  WITH RAISINS
   2B  CREAM OF WHEAT
   2C  CORN FLAKES
3  EGGS
   3A  SCRAMBLED
   3B  FRIED
       3B1  SUNNY-SIDE-UP
       3B2  OVER-EASY
   3C  BOILED
4  BEVERAGE
   4A  HOT CHOCOLATE
   4B  TEA
       4B1  WITH LEMON
       4B2  WITH SUGAR AND CREAM
   4C  COFFEE
```

FIGURE 2 - THE PROTOCOL

SUBSTITUTE WORD IN STATEMENT

Before we end this lesson, I'd like to show you how to change the content of statements which have already been inserted in your file.

You've learned how to correct errors by deleting an entire statement and inserting a new one in its place. The Substitute command may be used to change just a word or even a single character at a time. For example, here's how you can change a word in statement 2A1 so that it reads WITH DATES instead of WITH RAISINS.

```
EDIT C: (Su )bstitute C: (W )ord in  C: (S )tatement at  A: (2A1
)(<CR>);
 <New WORD> T: (DATES)(<CR>);
 <Old WORD> T: (RAISINS)(<CR>);
Finished? Y/N: (Y) OK: (<CR>);
Substitutions made: 1
EDIT C: (Do )ne OK: (<CR>);
```

NLS-SCHOLAR performs a command for the student much as a human tutor would by leaning over the student and typing the command on the student's terminal. Parentheses enclose what the "tutor" actually "types" separating it from the echoes and responses printed by NLS.
<CR> is just a symbol to denote that the "tutor" depressed the RETURN key. The semicolon is NLS's echo to a <CR>.

You can see that I've specified that I want to substitute one word in statement 2A1 for another; the new word DATES is to be substituted for the old word RAISINS.
(There is no restriction that the items substituted for one another be of the same length.)

To do this I typed only 'Su W S 2A1 <CR>' to produce the first line; I typed 'DATES <CR>' and 'RAISINS <CR>' to produce the second and third. The remaining characters are all printed by NLS.

NLS asks 'Finished?' since more than one substitution can be made at a time. If I had typed 'N' for no instead of 'Y' for yes, it would have prompted for another <New WORD> and <Old WORD> pair.

After I typed 'Y' NLS asks for further confirmation with OK:
to which I responded with a <CR>.

When the command has been executed, NLS tells how many substitutions have been made. If there had been two instances of RAISINS in statement 2A1, both would have been changed to read DATES.

Statement 2A1 is now the current statement. Please print it.

Student typing is underlined.

TASK 41

EDIT C: /
 ==>WITH DAT
EDIT C: ↑
 2A OATMEAL

The student confuses the \ command, which prints the current statement, with the / command that shows the current position of the control marker. Realizing his mistake, he fumbles with the other one-character commands he can remember. First he tries ↑ which prints the previous statement (2A), and makes it become the current statement.

EDIT C: _%

 2A1 WITH DATES
EDIT C: Done OK: _;

Next the student depresses the LINE FEED key which echoes as %, giving him the next statement (2A1), making it current.

You printed too many statements.
This is what I wanted you to print:
 2A1 WITH DATES

The task is evaluated.

Please redo just the printing
part of this task. Your file is unchanged.

NLS-SCHOLAR is really determined to get the student to review how to print the current statement.

continuation of TASK 41

EDIT C: *What command prints the current statement?*

 THE USER PRINTS THE CURRENT STATEMENT USING THE
\ COMMAND.

Failing to remember the appropriate one-character command, the student gives up and asks for help.
Questions can be asked anytime it is the student's turn to type.
NLS-SCHOLAR answers this question by retrieving information from its own internal, descriptive data base.

continuation of TASK 41

EDIT C: \
 2A1 WITH DATES
EDIT C: Done OK:_;

QUESTIONS?

** *The student chooses to ask no questions at this time.*

Try the Substitute command yourself by substituting the word OAT for CORN in the statement containing "CORN FLAKES".

TASK 42

EDIT C: *What statement contains "CORN"?*

THE STATEMENT CONTAINING "CORN" IS:
 2C CORN FLAKES

The student wants to know what the correct statement number is for him to type. The system obtains the answer by actually looking into the student's work space -- the answer is not stored (it cannot be!) in the internal data base.
The system interprets correctly most of the common paraphrases of this type of question, e.g.
"What is the statement containing..."
"What statement contains..."
"Where is...", etc.

continuation of TASK 42

EDIT C: Substitute C: Word in C: Statement at A: 2C;
 <New WORD> T: CORN;
 <Old WORD> T: OAT;
Finished? Y/N: Y OK:_;
Substitutions made: 0
EDIT C: Done OK:_;

The student switches around the new and the old words, so no change is made.

You should have altered the file so that this part of it
    2B CREAM OF WHEAT
    2C CORN FLAKES
 3 . . .

would look like this:
    2B CREAM OF WHEAT
    2C OAT FLAKES
 3 . . .

NLS-SCHOLAR shows the student what the result of this command should have been if he had performed the task correctly as opposed to the result which he obtained. With this information, the student can "debug" his performance of the task.

But you did not alter the file.

Please try to repair your mistake. The file is as you left it.

continuation of TASK 42

```
EDIT C: Substitute C: Word in  C: Statement at  A: 2C;
 <New WORD> T: OAT;
 <Old WORD> T: CORN;
Finished? Y/N: Y  OK:_;
Substitutions made: 1
EDIT C: Done OK:_;
```

TASK 43

```
EDIT C: Print OK:/C:_;
       2C OAT FLAKES
    3 EGGS
       3A SCRAMBLED
       3B FRIED
          3B1 SUNNY-SIDE-UP
          3B2 OVER-EASY
       3C BOILED
    4 BEVERAGE
       4A HOT CHOCOLATE
       4B TEA
          4B1 WITH LEMON
          4B2 WITH SUGAR AND CREAM
       4C COFFEE
EDIT C: Done OK:_;
```

NLS allows the student to type as little or as much of each command as he likes, providing it is unambiguously denoted. Here he chooses to type all of it since the command is so short.

```
You printed too many statements.
This is what I wanted you to print:
 2C OAT FLAKES
```

Please try to print what I wanted. The file is as you left it.

continuation of TASK 43

```
EDIT C: \
 2C OAT FLAKES
EDIT C: Done OK:_;
```

Good. The noun 'WORD' refers to the fact that whole words are involved, not just parts of words; but it does not imply that a single word only may be provided as either the <New WORD> or the the <Old WORD>. Use the Substitute command to change statement 2A1 again, this time to read WITH BROWN SUGAR AND CREAM instead of WITH DATES and print your result.

TASK 44

EDIT C: Substitute C: Word in C: Statement at A: 2A1;
<New WORD> T: WITH BROWN SUGAR AND CREAM;
<Old WORD> T: DATES;
Finished? Y/N: Y OK:_;
Substitutions made: 1
EDIT C: Done OK:_;

I wanted you to change
this part of your file:
    2A OATMEAL
       2A1 WITH DATES
    2B . . .

into this:
    2A OATMEAL
       2A1 WITH BROWN SUGAR AND CREAM
    2B . . .

But you changed this part of it:
    2A OATMEAL
       2A1 WITH DATES
    2B . . .

into this:
    2A OATMEAL
       2A1 WITH WITH BROWN SUGAR AND CREAM
    2B . . .

I have restored the file as it was
at the beginning of this task. Please start over.

TASK 44

EDIT C: *How does one do this?* *The student is lost and asks for help.*

THE USER TYPES 'SUBSTITUTE ', FOLLOWED BY 'WORD ',
'STATEMENT ', '2A1', <CR>, 'BROWN SUGAR AND CREAM',
<CR>,
'DATES', <CR>, 'Y', <CR>, '\', A <SPACE>, 'DONE ',
AND <CR>.

NLS-SCHOLAR can show him what to do, as well as...

continuation of TASK 44

EDIT C: *Please do it for me*

...actually do it for him!

```
EDIT C: (Substitute ) C: (Word ) in  C: (Statement ) at  A: (2A1
)(<CR>);
 <New WORD> T: (BROWN SUGAR AND CREAM)(<CR>);
 <Old WORD> T: (DATES)(<CR>);
Finished? Y/N: (Y)  OK: (<CR>);
Substitutions made: 1
EDIT C: (\ )
 2A1 WITH BROWN SUGAR AND CREAM
EDIT C: (Done ) OK: (<CR>);
```

QUESTIONS?

**

Please print the file starting with statement 1 so you can see the effect of the changes we have made.

TASK 46

EDIT C: *Tell me how to print the file.*

```
 THE USER PRINTS AN NLS FILE FROM THE CURRENT STATEMENT
TO THE END OF AN NLS FILE USING THE PRINT COMMAND,
 PROCEDURE: HE TYPES 'PRINT ' FOLLOWED BY <CR>.
```

An alternative to asking how to do a specific task (in this case TASK 46) is for the student to find his own way by asking more general kinds of questions.

continuation of TASK 46

EDIT C: *what does current statement mean?*

```
 THE CURRENT STATEMENT IS THE STATEMENT AT
 WHICH THE CM IS CURRENTLY POINTING.
```

continuation of TASK 46

EDIT C: *Where does the CM point to now?*

THE CURRENT POSITION OF THE CM IS:
 = 3B + 1

continuation of TASK 46

EDIT C: *How do I move the control marker?*

THE USER MOVES THE CM USING A CM MOVING COMMAND.

continuation of TASK 46

EDIT C: *What are the CM moving commands?*

EXAMPLES OF A CM MOVING COMMAND ARE THE JUMP COMMAND,
THE < COMMAND, AND THE > COMMAND.

continuation of TASK 46

EDIT C: *How do I jump to statement 1?*

THE USER MOVES THE CM USING THE JUMP COMMAND,
PROCEDURE: HE TYPES 'JUMP ', FOLLOWED BY '1', AND
<CR>.

The student has asked many questions to re-familiarize himself with the information necessary to perform TASK 46. He now feels able to proceed to do it.

continuation of TASK 46

```
EDIT C: Jump to A: 1;
EDIT C: Print OK:/C:_;
    1 JUICE
       1A ORANGE
       1B GRAPEFRUIT
    2 CEREAL
       2A OATMEAL
          2A1 WITH BROWN SUGAR AND CREAM
       2B CREAM OF WHEAT
       2C OAT FLAKES
    3 EGGS
       3A SCRAMBLED
       3B FRIED
          3B1 SUNNY-SIDE-UP
          3B2 OVER-EASY
       3C BOILED
    4 BEVERAGE
       4A HOT CHOCOLATE
       4B TEA
          4B1 WITH LEMON
          4B2 WITH SUGAR AND CREAM
       4C COFFEE
EDIT C: Done OK:_;
```

QUESTIONS?

How do I print just branch 3?

```
 THE USER PRINTS BRANCH 3 USING THE PRINT COMMAND,
 PROCEDURE: HE TYPES 'PRINT ', FOLLOWED BY
'BRANCH ', '3', <CR>, VIEWSPECS, AND <CR>.
```

Answer obtained by simple deductive inferencing.

**

REVIEW OF LESSON TWO

Lesson Two has introduced the following commands:

```
Insert command
Create command
Substitute command for words in statements
                   for characters in statements
                   for characters in branches
```

QUESTIONS?

NLS

The student can summon NLS in order to practice (or just to play) without fear of messing up his work.

...USING A DISPOSABLE COPY OF YOUR CURRENT FILE

```
EDIT C:  Print  OK:/C:  Statement at A: 2B;
 V: _;
2B  CREAM OF WHEAT
EDIT C: Substitute C: Character in  C: Statement at  A: 2B
;
 <New CHARACTER> T: U;
 <Old CHARACTER> T: EA;
Finished? Y/N: Y  OK:_;
Substitutions made: 2
```

The student replaced the characters EA for U.

```
EDIT C: \
2B CRUM OF WHUT
EDIT C:  Print  OK:/C:  Branch at  A: "EGGS";
 V:_;
  3  EGGS
     3A  SCRAMBLED
     3B  FRIED
         3B1  SUNNY-SIDE-UP
         3B2  OVER-EASY
     3C  BOILED
EDIT C: Substitute C: Character in  C: Branch at  A: 3;
 <New CHARACTER> T: *;
 <Old CHARACTER> T: -;
Finished? Y/N: Y  OK:_;
Substitutions made: 3
EDIT C: Print OK:/C:  C: Branch at A: 3B;
 V:_;
 3B FRIED
    3B1 SUNNY*SIDE*UP
    3B2 OVER*EASY
EDIT C: Done OK: ;
```

...THROWING AWAY THIS COPY

The student leaves NLS returning to the 'QUESTIONS?' level. The "Transparent overlay" on which he has scribbled disappears without trace.

Print branch 3B, please

Commands can be issued in natural language.

```
...USING A DISPOSABLE COPY OF YOUR CURRENT FILE

EDIT C: (Print ) OK:/C: (Branch ) A: (3B) (<CR>);
V: (<CR>);
  3B FRIED
     3B1 SUNNY-SIDE-UP
     3B2 OVER-EASY
EDIT C: (Done ) OK: (<CR>);

...THROWING AWAY THIS COPY
```

The Tutor demonstrates how to do it. Notice that the file is in its original state.

**

It has also introduced the following concepts:

echoing
field
level
level zero
L: prompt
level adjustment
   up (u)
   down (d)
   same (<CR>)

repeat mode
<CTRL-B>
question mark facility

QUESTIONS?

What can I type after L:?

THE USER TYPES <CR>, 'u', 'd', OR A COMBINATION OF 'u' AND 'd' AFTER THE L: PROMPT.

What statements are at level 2?

THE STATEMENTS AT LEVEL 2 ARE:
1A 1B 2A 2B 2C 3A 3B 3C 4A 4B 4C

What would be the level of statement 14ac3?

THE LEVEL OF STATEMENT 14AC3 IS:
3

How does it work

Much of NLS-SCHOLAR's knowledge is derived from stored data and from a set of built in routines that manipulate and retrieve those data in response to queries. The data base is a semantic network of descriptive information that is represented in attribute-value format. It contains descriptions of actions and their purposes, descriptions of the procedures necessary to accomplish these actions, and descriptions of their effects and consequences. For example, the semantic network contains a representation of the description of the purpose of, and the procedure for issuing, the delete command. An English rendition of the this attribute-value representation would be: "The purpose of the delete command is to delete a structure unit", and "The procedure (for deleting a structure unit) is for the user to type the word DELETE, followed by the name of a structure unit, its address, and two carriage returns." The semantic network also contains many other kinds of attributes, among them the definition of concepts, and the interrelationships between concepts such as "A statement is an instance of (or a name of) a structure unit."

The retrieval routines, initiated by a query, search the semantic network for information relevant to the query. For example, if a student wants to know what the delete command does, his question would translate into a query that

would essentially mean: "Find the purpose of the delete command". The retrieval routines would attempt several different matching procedures that would finally yield: "The purpose of the delete command is to delete a structure unit."

The retrieval process is assisted by built-in "reasoning" strategies that are called upon when the matching procedures fail. In fact, in many cases the desired information is not stored specifically as demanded by the query, but may be inferred from available information. For example, if the query were for the procedure for deleting a statement, our matching procedures would fail. However, the system would still be able to derive an answer via simple deductive inference; it knows that a statement is a kind of structure unit, and it knows how to delete a structure unit, therefore the procedure is to "type DELETE, followed by STATEMENT, etc."

It is important to observe the introspective character of this form of cognition. We have a) a data base that is static, internal, and is made out of symbols, and b) a set of built-in inference strategies and retrieval routines that operate on those static, internal, symbolic representations.

Inferencing and retrieval mechanisms such as the ones just described are the seat of the abstract "thinking" abilities of NLS-SCHOLAR. As such, they are not yet very

powerful, and much can be (and will be) done to improve them.* However, it is important to stress here that there is more to "intelligence" than powerful manipulation of symbols.

People´s intelligent behavior is not based solely on internal representations and conceptualizations and their attendant reasoning processes. A person´s data base is not only memory, and his "retrieval routines" are not solely introspective: he uses the world as a data base and his senses to retrieve information from it. I don´t need to have in my head a representation of what is behind my chair; if I need to know, I can just turn around, look, and see!

Due to the fact that NLS-SCHOLAR deals with a "world" (NLS´s world) with which it shares much of its own being, (i.e. it is a program that deals with the use of another computer program) it was relatively easy to endow it with some of this latter kind of "intelligence". For example, to make NLS-SCHOLAR "aware" of the state of the student´s work, all we had to do was design the system so that it could use NLS as a sort of sensor. Thus when the student, lost in thought, asks a question about his work space (such as ´What was the address of that statement that contained "DESSERT"?´ or simply "Where is "DESSERT"?´) NLS-SCHOLAR manufactures an

---

*Much work has been done on this problem in the SCHOLAR system that deals with the geography of South America [9].

apposite command, has it executed (invisibly) by LISP-NLS (see below), and uses the result to construct an answer. Moreover, NLS-SCHOLAR is designed to use LISP-NLS as its seat of pragmatic inferential knowledge. For example, sometimes it is easier to obtain an answer by actually "doing" and then "looking and seeing", rather than by deducing the answer via logical inferences. This method is very powerful -- sometimes it is not just easier to do than to deduce: it is the only way we know of deriving an answer. A new breed of "intelligent" CAI systems based on this approach has been pioneered by Brown and his SOPHIE system [6,7].

NLS-SCHOLAR can also combine the two forms of knowledge. That is, it can use its semantic network and reasoning routines to infer a procedure (such as how to delete a statement) and then use this procedure to synthesize an NLS command and have it executed. Thus NLS-SCHOLAR can both describe and do things.

In the above discussion we carefully avoided asserting that NLS-SCHOLAR actually uses the real NLS system. For a number of reasons, we preferred to write our own version of NLS in INTERLISP [10], and to wait until NLS-SCHOLAR reaches a stable state before interfacing it with the real NLS. We have taken elaborate precautions to ensure that this switch can be done with a minimum of re-programming. All exchanges

between NLS-SCHOLAR and our LISP-NLS take place at the surface language level (the system does manufacture NLS-executable command strings that are then executed by LISP-NLS) and we have consistently resisted the temptation to short-cut this path.

SECTION II - NLS-SCHOLAR AS A TUTOR

Introduction

Having evolved from SCHOLAR [2,3], NLS-SCHOLAR is an interactive, mixed-initiative system that is capable of answering freely interspersed questions posed by a student in the course of a tutorial session. However, the differences in subject matter (text editing, computer based systems vs. geography of South America) and in aim (learning how to use a system vs. learning descriptions and names) are of such magnitude that NLS-SCHOLAR and regular SCHOLAR differ substantially in a number of important ways.

Consider first the differences in subject matter. Most people know the fundamental concepts and relations of geography, so that teaching the geography of South America doesn't have to start by introducing the concepts of country, capital, government, etc., and the relations between them, e.g. that countries have capitals, that governments reside in capitals. Rather, the instantiation of these relationships can be taught right away, e.g. that Colombia is a country in South America, and that its capital is Bogota. People's common knowledge of geography also enables them to ask meaningful and instructive questions from the start. Few people, however, know the fundamental concepts, relations, and operations that characterize the

use of a computer based text editing system, and they cannot learn very much about it by asking questions because they do not know what to ask for or how to ask for what they want to learn. Consequently, teaching must begin at a more basic level.

Instructive interactions must be based on an underlying conceptual structure that is common to the tutor and the student. If this underlying structure is rich (as in the case of geography), teaching is simple, and learning can benefit considerably from the student's being able to ask meaningful questions from the start. If the structure is shallow (as in the case of text editing systems) it must be built up before teaching can go very far. Therefore, one of the main goals of our work in NLS-SCHOLAR was to design and implement a tutorial mode especially adapted to this purpose.

Consider now the differences in aim. Most people can learn the geography of a region without much manipulation of the new facts that they learn. These facts sort of fit in fixed slots that are there beforehand and that represent well understood concepts. Few people, however, can really learn to edit text without practicing, that is without being able to perform editing operations and without being able to ask questions about the state of their work. Therefore, the other main goal of our work was to develop the means to

couple closely the "NLS world" with NLS-SCHOLAR, so that the student could be put in contact with NLS, while NLS-SCHOLAR, overseeing all this, could bring about SCHOLAR-like abilities to help the student when needed.

In what follows we describe how NLS-SCHOLAR teaches the fundamental concepts underlying text editing with NLS, and how it interfaces with NLS so that students can practice what they learn while remaining under tutorial supervision.

Teaching NLS fundamentals: The Primer

As discussed above, teaching people how to use a text editing system is entirely different from teaching them about the geography of a region. Therefore, for a SCHOLAR system to teach NLS effectively, a new set of tutorial strategies had to be developed in order to cope with the more basic concepts that must be introduced to the student.

Following a now well established path for developing these strategies [11], we set out first to find out how human tutors teach NLS and what are the most important and effective methods that good teachers use. We first studied the course offered at BBN by members of the Augmentation Research Center, and one of us (Laura Gould), having had considerable experience in teaching the use of computers to Humanities students, undertook to teach NLS to a small number of students (members of BBN's secretarial staff). An

analysis of the protocols of the teaching sessions pointed to several problem areas.

The difficulties of teaching NLS concepts solely by symbolic and formal descriptions can be appreciated in the following example concerning the way NLS files are organized and function.

Consider this portion of the DINNER file:

```
2  ENTREE
   2A  FRIED CHICKEN
   2B  PRIME RIBS
   2C  SALMON
     2C1  WITH CREAM SAUCE
   2D  SCALLOPS
     2D1  BROILED
     2D2  FRIED
```

The structure of NLS files is such that statement numbers represent slots or shelves that are provided by the system. If we remove a statement, another statement which follows it may be "promoted" to take its place, causing a reassignment of statement numbers. For example, after deleting the statement containing the PRIME RIBS, the file would be left as:

```
2  ENTREE
   2A  FRIED CHICKEN
   2B  SALMON
     2B1  WITH CREAM SAUCE
   2C  SCALLOPS
     2C1  BROILED
     2C2  FRIED
```

This action and its effects can undoubtedly be described formally without referring to any actual file. But how much simpler it is to do it by way of an example!

The main conclusion that we extracted [12] was that while effective teaching still depended on describing facts, actions, purposes, procedures, etc. symbolically, the most effective elements of the teaching situation were the ostensive ones, namely:

1) teaching by letting the students do things by themselves and helping them correct their mistakes.
2) teaching by way of examples
3) teaching by demonstrating actions (the tutor typing commands for the student, for example, when a complicated new command is being introduced or when the student is unable to proceed)

The Primer is like a scenario for the form NLS-SCHOLAR adopts when in tutorial mode. It is the consequence of skillfully organizing, segmenting, presenting and sequencing knowledge about NLS in a manner that results in easy and comfortable learning. (For a complete version of the Primer, see the Appendix.)

In tutorial mode, NLS-SCHOLAR consists of an agenda-driven sequence of tutorial units. The elements of these tutorial units are:

a) delivering information

b) asking questions of the student

c) showing examples

d) demonstrating actions

e) requesting the student to perform tasks and exercises, evaluating them, and making the appropriate comments to the student

f) pausing to answer questions from the student

Elements a) and f) are always present.

The way things work is as follows. NLS-SCHOLAR presents exposition, embedded in which is a series of tasks. Fairly frequently, the system stops to ask whether there are any questions, by typing QUESTIONS? in the margin and then printing an asterisk on the next line. If the student has no questions he types an asterisk followed by a <CR> and the exposition continues. If he has a question he type it directly following the "*" and terminates it with another "*" and a carriage return (<CR>) in typical SCHOLAR fashion. When the question has been answered, NLS-SCHOLAR prints another "*" in the margin indicating that it expects another question. If the student has no more questions, he types *<CR> and NLS-SCHOLAR proceeds.

Whenever a task is proposed, NLS-SCHOLAR puts the student in touch with NLS. This causes the herald EDIT and

the prompt C: to appear as EDIT C: in the left margin. The student can then type one of four things:

(1) an NLS command term

(2) a "?" to obtain a list of all command terms which are possible at that point

(3) a "*" to indicate that he wants to ask a question

(4) "DONE <CR>" to indicate that he has completed the task and wishes to have it evaluated

If he does (1) his actions will be stored for later evaluation. When his command is terminated, a new EDIT C: will appear in the margin.

If he does (2) a list of possible command terms will be printed. He should then type one of them and proceed with his command.

If he does (3) his question will be answered and a new EDIT C: will appear.

If he does (4) his performance of the task will be evaluated. If he has done the task correctly he will be praised and the exposition will continue. If he has done the task incorrectly his mistake will be pointed out to him, his file restored to its form before the task was initiated and he will be asked to do it again. He may ask the system to show him how to do it, or even ask the system to do it

for him if he is in real trouble.

Occasionally NLS-SCHOLAR will ask a question of the student. At such a point, a "*" is printed in the left margin, NLS-SCHOLAR waits for the student to answer the question, and then evaluates his answer.

Endowing NLS-SCHOLAR with 'awareness'

In order to make NLS-SCHOLAR 'aware' of what a user does with NLS, we had to develop a coupling that enabled NLS-SCHOLAR to use NLS to 'sense' the state of a user's file. This coupling constitutes an exceedingly powerful tool. First, observe that it makes it possible for the student to ask questions not only about descriptions, definitions, procedures, etc. (such as "What is a prompt," "What does viewspec n do," or "How do I delete a statement") but also about the current state of the student's work (such as "What is the content of statement 3A", or "Where is the Cm now" or "Print the current statement" all relative to the present state of the student's file). Thus, in addition to searching for answers in a semantic network in the "standard" SCHOLAR way, we gain the ability of interrogating the NLS world as well. Second, this coupling provides an easy way of performing a type of inference that would be very hard to perform deductively. Suppose a student asked 'If I deleted statement 3A1, what would then be the number

of the statement containing "TOMATO"?' Finding the answer by deductive reasoning is possible but difficult. Obtaining the answer by "sotto voce" deleting statement 3A1 and then seeing where the "TOMATO" statement ends up is much easier and very powerful. In Third, it becomes possible to evaluate easily a student's solution to a proposed task -- all the system has to do is to have available the correct sequence of commands for the task, perform them on a fresh copy of the current file, and then compare the results (in terms of the state of this new NLS file) with the student's file.

The LISP-NLS system

In order for NLS-SCHOLAR to teach NLS ostensively in the manner we have described, and in order for it to answer questions about the current state of the student's work, it is clear that NLS itself must be incorporated and interfaced with NLS-SCHOLAR. However, although using the real NLS for this purpose was entirely feasible (everything is on TENEX), we decided instead to implement the EDIT subsystem of NLS in INTERLISP. The reasons for this early decision were manyfold:

a) NLS was undergoing changes (it still is)

b) building a communication interface would have consumed a larger fraction of our limited funds than implementing our own LISP-NLS

c) the real NLS is a very complex system and we wanted to test the feasibility of our approach in an environment

we understood well

d) since NLS-SCHOLAR is written in INTERLISP, inter-process communication and control would be facilitated

The results were very beneficial. As it turned out, it was not only simple indeed to make NLS-SCHOLAR talk to LISP-NLS, but we learned a great deal from designing it such that interfacing NLS-SCHOLAR with the real NLS will require a minimum of re-programming. We fully realize that if our system is to attain the degree of operational usefulness it is capable of, it will have to be within the context of normal usage of the real NLS. This we expect to accomplish in the near future.

LISP-NLS is capable of performing almost all of the commands in the editing subsystem of NLS, and to users of NLS-SCHOLAR it looks exactly like the real thing. Rather than attempting to describe its inner workings, let us instead describe it operationally, from the point of view of performing the functions required by NLS-SCHOLAR.

The top function of LISP-NLS is called NLSPARSE and it takes as an argument a single character. When a command is being issued for LISP-NLS to perform, the command string is fed to it character by character. NLSPARSE digests the character and returns as a value a list of three elements. The first element is a parameter used to determine what to

do next (feed the next character, signal that the command has been completed, etc.). The second element is the "echo", i.e. what NLS normally prints when one types a character (the character itself plus whatever prompts and heralds may be required at the time). For example, in expert mode, typing "I" as the first character of a command results in NLS echoing "Insert C:". The third element appears only after the character that completes a command has been fed to NLSPARSE and it contains the response (what NLS normally prints as a result of executing the command) plus a wealth of data about the state of the NLS file as a result of having performed the command. These data are: the parsed command string, a representation of the file's structure, the position of the control marker, the state of the viewspecs, and a list of what was printed by means of any of the various print commands available in NLS. These data are used by the task evaluation machinery to figure out whether or not a student performed a task correctly.

Observe that the passing of characters and the confinement of output to "echoes" and responses makes it possible to use LISP-NLS very flexibly. Input characters, for example, can be fed to it as one normally would to NLS, namely by typing them on a terminal. Alternatively, they may be fed to LISP-NLS by retrieving them from the data base (having commands stored under each task in the data base makes it possible for NLS-SCHOLAR to simulate the typing of

commands by a human tutor).

Echoes and responses can be similarly controlled. For example, when the Question Answering system synthesizes a command to LISP-NLS, echoes are not used at all and responses are not printed directly but are handed back to the Question Answering system to be used in constructing a response to the student.

In addition, a context manipulation machinery allows the storing and retrieving of environments, and the creation of new ones. This is necessary when, for example, the student asks a question that requires the Q/A system to synthesize a command that could alter the state of his file. For example, if the student asked "What is the content of statement 3?" and the control marker were positioned at statement 1, the Q/A system would have to synthesize a command that would result in the CM being positioned to statement 3 in order to answer the question. However, all evidence of this command's execution must be removed -- in particular the control marker must be repositioned to statement 1 -- or the student will be confused by the state of his file which has been manipulated without his knowledge. Saving the student's environment, performing invisible commands on a disposable copy of it and restoring the environment afterwards, solves the problem. Other examples of context manipulation can be seen in Section IV,

in the description of the task monitoring machinery.

SECTION III - STUDENT/QA AND TUTOR/QA SYSTEMS

Questions and Answers: an Overview

In the course of a lesson, or in the course of their own independent work, users of NLS-SCHOLAR can ask questions for the system to answer. In the course of a lesson the system also generates questions for the student to answer and then evaluates those answers. The system that answers student-generated questions is called Student/QA, while the system that generates questions and evaluates a student's answers is called Tutor/QA. For consistency, Tutor/QA must be able to generate the same set of questions that Student/QA can answer. This enables Tutor/QA to perform answer evaluation by simply passing off the question it generates to Student/QA to derive the correct answer. This correct answer can then be compared to the student's answer and appropriate action taken.

Because both the Student/QA and Tutor/QA systems involve the same set of requests, we have designed both to use the same representation of the meaning of a request. In this way, Student/QA responds to a student's request by parsing it into this representation, or "semantic form". This semantic form is just a LISP procedure which, when executed, derives the answer. "Semantic forms" are also used by Tutor/QA to produce a question to present to the

student (i.e. one which if the student had asked it would have parsed into the identical form). At the same time it evaluates this semantic form to derive the answer and compares it with the student's answer. Both what these forms look like, how they are derived, and what it means to evaluate them and get an answer will become clearer in the following discussion of the two systems and their interplay.

Student/QA

Student/QA is responsible for answering students' questions about the EDIT subsystem of NLS and about the current state of his NLS file. It derives its answers from two sources of information: a data base containing static descriptions, and NLS itself (actually LISP-NLS).

Like previous SCHOLAR systems, NLS-SCHOLAR has a data base organized as a semantic network containing definitions and examples of concepts, descriptions of procedures, etc. This semantic network represents time-invariant factual information about NLS and has been structured so as to facilitate the kinds of inferences required for answering questions such as:

```
WHAT IS A HERALD?
GIVE ME SOME EXAMPLES OF STRUCTURE UNITS.
HOW DO I PRINT THE NEXT STATEMENT?
HOW DO I DELETE THE LAST CHARACTER THAT I'VE TYPED?
```

But in order for NLS-SCHOLAR to respond to some of the real

needs of a student engaged in learning NLS, it becomes necessary for Student/QA to handle questions relating to what the student is doing, i.e. the state of the student's work with his NLS file. A few questions of this type are:

WHAT IS THE CONTENT OF STATEMENT 3A?
WHERE ARE THE "SCALLOPS" NOW?
WHAT STATEMENTS ARE AT LEVEL 3?

None of these questions can be answered with the static information in the semantic network (although this static information is sometimes used to synthesize a plan for obtaining the answer). The semantic interpretation of this type of question instead results in a call to LISP-NLS to perform a series of synthesized NLS commands (executed invisibly to the student). This means that there must be a system (discussed in Section V) which saves the student's environment, performs the synthesized commands, restores the student's environment, and hands back the result of executing these commands to Student/QA which in turn responds to the student.

The Parser:

The NLS-SCHOLAR parser performs a top-down, semantically directed case analysis of a sentence based on the grammar described in BNF form in Figure 3. This method is much like that used in the SOPHIE system [6,7]. The parser produces a semantic form that contains information similar to that derived from the "English Comprehender" of

NET-SCHOLAR [5], including the assignment of case relationships existing between the main verb and the noun phrases of the input sentence. In addition, this method determines the general category that the request falls into (a request for a definition, a request for a procedure, a request for the address of some word in the current file, etc.) For example, for the request:

HOW DO I DELETE STATEMENT 2A?

the semantic form would look like:

```
(QFIND/PROCEDURE ((AGENT USER)
                  (VERB DELETE)
                  (OBJ STATEMENT (ADDR 2A))))
```

The semantic form of all requests is a LISP function which can be EVALuated, (that is, QFIND/PROCEDURE is a LISP function to find a procedure in the data base; it takes a case-structure parse as its input, retrieves the correct information from the data base and calls the Output package to output the answer in sentence format). To give a better idea of this process, we will follow through parsing, retrieval and output for the request:

HOW DO I DELETE STATEMENT 2A?

Parsing in Detail:

The parser first does a pre-scan of the sentence. This pre-scan does spelling correction (using the routines from the BBN INTERLISP DWIM facility [13]), abbreviation

checking, and compound word checking, making words like "DELETE COMMAND" into a single concept "DELETE\COMMAND". This prescan rewrites the input as "HOW\DO\I DELETE STATEMENT 2A".

Parsing proper begins at this point. The description will be best understood by following it through with the BNF description of the grammar in Figure 3. In fact, the parsing algorithm is almost isomorphic to the grammar, and many of the LISP functions that make up the parser have the same names as the elements of the grammar.

The top-level function <REQUEST> first checks to see if the sentence is a request for the definition of something. In our case it isn't. It continues its depth-first search until it reaches <PROCEDURE/REQ> which first checks to see if the sentence begins with the concept "HOW\DO\I". This succeeds and RESULT, a global variable that keeps track of the parse, is set to:

(QFIND/PROCEDURE (AGENT USER))

"HOW\DO\I" is removed from the string. The parser then tries to locate an <ACTION/SPEC>, that is, a <VERB> plus any number of <OBJ>'s, with the remaining string "DELETE STATEMENT 2A".

FIGURE 3 -- A BNF DESCRIPTION OF THE GRAMMAR

```
<REQUEST>:= <DEFINE/REQ>
            <WHATIS/REQ>
            <CONTENT/REQ>
            <PARTS-IN-PART/REQ>
            <PARTS-IN-LEVEL/REQ>
            <PROCEDURE/REQ>
            <INSTR/REQ>
            <POSITION/REQ>
            <NLS/ACTION/REQ>

<DEFINE/REQ>:= DEFINE <NOUN>
               WHAT DOES <NOUN> MEAN
               WHAT DOES <NOUN> STAND FOR
               WHAT DOES <NOUN> DO

<WHATIS/REQ>:= WHAT IS THE PURPOSE OF <NOUN>
               WHAT IS THE CONTENT OF <STR+ADDR>
               WHAT IS THE LEVEL OF <STR+ADDR>
               WHAT IS THE ADDRESS OF <STR+ADDR>
               WHAT ARE EXAMPLES OF <NOUN>
               WHAT IS THE DEFINITION OF <NOUN>
               WHAT IS <CURRENT/PART>
               WHAT IS <STR+ADDR>
               WHAT ARE <NOUN>
               WHAT ARE <STRUCTURAL> AT <LEVEL/PART>
               WHAT ARE <STRUCTURAL> IN <FILE/PART>
               WHAT IS <NOUN>
               **ALSO 'TELL\ME, GIVE\ME, TELL\ME\ABOUT' IN
                PLACE OF 'WHAT IS'

<CONTENT/REQ>:= WHAT <STRUCTURAL> CONTAINS <STRING>

<PARTS-IN-PART/REQ>:= WHAT <STRUCTURAL> ARE IN <FILE/PART>

<PARTS-IN-LEVEL/REQ>:= WHAT <STRUCTURAL> ARE AT <LEVEL/PART>

<PROCEDURE/REQ>:= HOW\DO\I <ACTION/SPEC>
                  TELL\ME\HOW\TO <ACTION/SPEC>
                  TELL\ME\ABOUT <ACTION/SPEC>

<INSTR/REQ>:= WHAT [NLS\COMMAND] <ACTION/SPEC>

<POSITION/REQ>:= WHERE AM I
                 WHERE IS THE CM
                 WHERE IS <STR+ADDR>

<NLS/ACTION/REQ>:= <ACTION/SPEC>
                   DO IT
                   DO <TASK>
```

```
                    WHAT HAPPENED
                    WHAT IS WRONG
                    HOW\DO\I DO THIS TASK
                    HOW\DO\I DO <TASK>
                    SHOW\ME\HOW\TO DO THIS

<TASK>:= TASK <NUMBER>

<NUMBER>:= Ø ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9

<ACTION/SPEC>:= <VERB> [<OBJ>]

<VERB>:= ANY WORD IN THE DATA BASE WHOSE PART OF SPEECH
            INCLUDES "VERB"

<OBJ>:= <NOUN/PHRASE> [<OBJ>]
        <RELATIONAL> <NOUN/PHRASE> [<OBJ>]

<RELATIONAL>:= ABOVE ! AFTER ! AT ! BEFORE ! BELOW
               FOLLOW ! FOLLOWING ! FOR ! FROM
               IN ! NEXT\TO ! OF
               THROUGHOUT ! TO ! USING ! WITH
THE/BEGINNING/OF ! THE/END/OF ! FOLLOWING

<NOUN/PHRASE>:= <NOUN>
                <STR+ADDR>

<STR+ADDR>:= <FILE/PART>
             THE <STRUCTURAL> <STRING>
             THE <TEXTUAL> <STRING>
             <CURRENT/PART>
             <STRING>

<STRUCTURAL>:= STATEMENT ! BRANCH !  PLEX ! GROUP

<TEXTUAL>:= WORD ! CHARACTER ! VISIBLE ! INVISIBLE

<CURRENT\PART>:= THE CURRENT\STATEMENT
                 THE NEXT\STATEMENT
                 THE BACK\STATEMENT
                 THE CURRENT\VIEWSPECS
                 THE CURRENT\ADDRESS
                 THE CURRENT\STATEMENT\NUMBER
                 THE CURRENT\POSITION\OF\THE\CM
                 THE CURRENT\FILE

<FILE/PART>:= STATEMENT\Ø
              STATEMENT <ADDRESS>
              BRANCH <ADDRESS>
              PLEX <ADDRESS>
              GROUP <ADDRESS> <ADDRESS>
```

<ADDRESS>:= AN ATOM WHOSE FIRST CHARACTER IS A NUMBER

<LEVEL/PART>:= LEVEL <NUMBER>

<TASK>:= TASK <NUMBER>

<STRING>:= ACTUAL PIECE OF TEXT IN QUOTES ("")

<NOUN>:= ANY WORD IN THE DATA BASE WHOSE PART OF SPEECH
INCLUDES "NOUN"

<ACTION/SPEC> finds the verb "DELETE" and then succeeds in finding a sentence object that matches <FILE/PART> under <NOUN/PHRASE>. <ACTION/SPEC> appends to RESULT the expression (VERB DELETE) (OBJ STATEMENT (ADDR 2A)). Parsing is now completed having reached a terminal state in the grammar. The value for RESULT is:

```
(QFIND/PROCEDURE ((AGENT USER)
                  (VERB DELETE)
                  (OBJ STATEMENT (ADDR 2A))))
```

RESULT is now EVALuated retrieving the correct procedure from the data base and calling the Output package to construct the sentences to be typed out to the student.

There are objections to having a non-general parsing algorithm, but for NLS we believe that the pros outweigh the cons. This algorithm is fast* and it can be expanded (it has been changed already many times) to cover the types of questions we discover our students asking most often. If experience with the SOPHIE system, which is used for electronic troubleshooting, is a good indication of what we may expect, then we shall not run into too much difficulty with this kind of parser.

---

* A typical request parses in ms.

Retrieval:

As discussed previously, we are dealing with a new and quite interesting facet of knowledge: that of the interaction of static and dynamic information. When the student asks "WHAT DOES CTRL-X DO?" the answer is a static piece of knowledge retrievable from the semantic network. But when the student asks "WHAT IS THE CURRENT STATEMENT?" the answer will not be found in the semantic network; it depends on what the student is doing and must be extracted from his work space. When a request is found to require information about the state of a student's file, the necessary NLS commands are formulated by the top-level retrieval function; LISP-NLS is called to perform the commands and to return the result of performing them. In this case, the retrieval function QFIND/CONTENT requests LISP-NLS to perform the NLS command "\" which returns the contents of the current statement.

These two examples elucidate the need for two kinds of "data bases": a static semantic network and a dynamic NLS file. So far we have seen their use in separate and clearly distinguishable cases. However, when the student requests the Tutor to perform a specific NLS command for him (for example, he may say "PLEASE PRINT BRANCH 6A FOR ME") then the two data bases must interact in order to produce a response. To fulfill that request, Student/QA must first

find the procedure for printing branches. This procedure is very general and static so it is stored in the semantic network. Loosely speaking, it states that one should type "PRINT", followed by the name of the structure unit to be printed, its address, <CR>, viewspecs, and <CR>. To obtain an answer, Student/QA must use this general piece of knowledge as a plan to synthesize a legal NLS command. It must now "instantiate", according to the information supplied in the student's request, the name of the structure unit, its address, and the viewspecs. For this request, the NLS command formulated would be:

Print Branch 6A <CR> <CR>

Instantiation is made possible by having a set of instantiation variables containing the current instance of a number of generic concepts. This collection of instances is cleared before a question is asked, and is filled in (if required) during the parsing of the request. Before an answer is constructed or an NLS command synthesized, Student/QA checks to see if the instantiation variables were filled in during parsing, and if so, uses them in place of the generic terms.

In this example all the generic terms in the procedure had to be instantiated; that is, the generic term "structure unit" is the word "BRANCH", the address is "6A", etc. This is because without instantiating <u>all</u> these generic terms,

the NLS command would not be considered legal -- LISP-NLS would not be able to perform it. Sometimes instantiating specific terms for more general ones is not really critical, but is more a matter of producing a better response to a student's question. For example, statements, branches, groups and plexes are all instances of structure units. The procedure for, say, deleting a statement is not stored individually since the procedure is the same for deleting any structure unit. Only the general rule is stored. Because of this, instantiation of certain objects in the semantic network is preferable (but not essential) so that questions like "HOW DO I DELETE A STATEMENT" do not get answered "THE USER DELETES A STRUCTURE UNIT WITH THE DELETE COMMAND".

Output:

The Output package is essentially the same as that of NET-SCHOLAR with one important addition. We wished to allow items to be instantiated to produce a meaningful response or a "legal" NLS command. In the semantic network this shows up as a new structure made up of three elements: $INS, a variable, and a piece of regular SCHOLAR data base. (See Figure 4.)

When the Output system encounters a list beginning with $INS, like ($INS XADDSTR ADDRESS), it checks to see if the

second item, in this case the variable XADDSTR, has a value (set during the parse). If so it uses this value in constructing the answer. Otherwise it uses the third item, ADDRESS, a generic term like any regular piece of SCHOLAR data base.

For example, in the question "HOW DO I DELETE A STATEMENT?" the parser sets the variable XOBJ to STATEMENT, and the variable XOBJSTR to 'STATEMENT'. Retrieval finds the piece of data base answering the general question "HOW DO I DELETE A STRUCTURE UNIT?" which is the procedure listed in Figure 4. This is sent off to Output which creates a sentence with the appropriate instantiated elements. (XOBJ and XOBJSTR are instantiated: that is, their values, STATEMENT and 'STATEMENT' respectively, are used. No value for XADDSTR was assigned during the parse (the student didn't specify a specific address) so the third item in the $INS list, the generic term ADDRESS, is used. The response is:

THE USER DELETES A STATEMENT USING THE DELETE COMMAND.
PROCEDURE: HE TYPES 'DELETE', FOLLOWED BY 'STATEMENT', AN ADDRESS, <CR>, AND <CR>.

FIGURE 4 - DATA BASE ENTRY FOR "DELETE\COMMAND"

```
DELETE\COMMAND
 (PURPOSE (I 2)
  (DELETE
   NIL
    (AGENT NIL USER)
    (OBJ NIL ($INS XOBJ ($EOR STRUCTURE\UNIT STRING\UNIT)))
    (INSTR NIL DELETE\COMMAND)
    (PROCEDURE
     NIL
      (TYPE
       NIL
        (AGENT NIL USER)
        (OBJ
         NIL
          ($SEQ "DELETE "
                ($INS XOBJSTR
                      ($EOR (NAME NIL (OF NIL STRUCTURE\UNIT))
                            (NAME NIL (OF NIL STRING\UNIT))))
                ($INS XADDSTR ADDRESS)
                <CR> <CR>)))))
```

Tutor/QA

The Tutor/QA system was designed to make use of the same semantic form that the Student/QA system produces during a parse. This integration allows us to make use of Student/QA's retrieval functions to derive the correct answer to a Tutor-generated question so that this answer can be checked against the student's. This integration of both QA systems is illustrated in Figure 5. In this diagram the blocks represent the specialists some of which are shared among both systems. The arcs are labelled with both inputs to various blocks and their outputs. Also some tests are made explicit on the arcs, like whether it was the student or the Tutor who initiated the request.

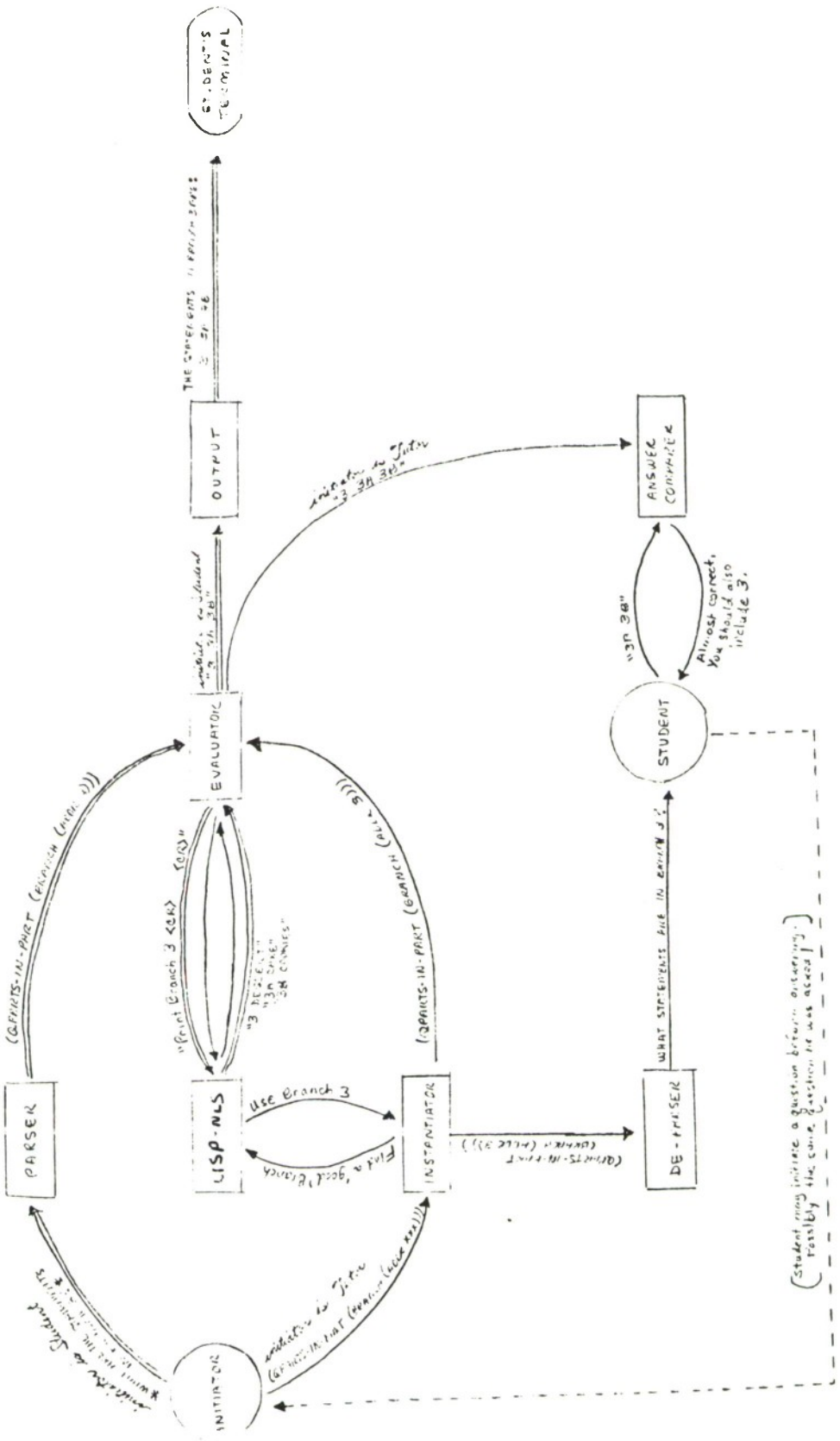

FIGURE 5 - INTEGRATION OF TUTOR/QA AND STUDENT/QA

The path from Initiator to Parser to Evaluator to Output is that of the Student/QA system, that is, the Initiator is the student. (This pathway is marked by double-line arcs.) Tutor/QA's integration of its pathways with this one will be made clearer in the following section.

Tutor/QA System's Organization:

This system is activated by a call to Tutor/QA to generate a suitable question to ask the student. The general type of question is designated by the agenda, and is represented as an incomplete semantic form exactly like that produced by the parse of a student's request in Student/QA but with several operands represented as variables whose values have yet to be filled in.

This form is handed to the Form-Completer who chooses "good" values for the variables in the incomplete semantic form. It often arrives at "good" choices by taking a look at the current NLS file with a call to LISP-NLS. Once a semantic form is complete (all variables filled in), then two activities take place simultaneously. One is a call to the Evaluator who evaluates the semantic form, i.e. retrieves the correct answer. The second is a call to the De-Parser, a specialist who takes the semantic form and, in a sense, de-parses it; it reverses the process done in Student/QA by going from the semantic form back to its

canonical surface representation. This surface representation (the question phrased in "English") is then presented to the student. The student's response to it is compared with the correct answer derived from the Evaluator. The Answer Comparer decides whether the responses are equivalent. It responds to the student appropriately and returns a message to Tutor/QA as to how the student did with this question. Tutor/QA can then decide whether it feels it should generate another question of this type (if the student did poorly), or whether it should return so that the lesson may continue. If the decision is made to go on, Tutor/QA exits returning control over to the system executive. If it decides to continue asking similar questions until the student has gained sufficient familiarity with the concept that it is trying to get across, Tutor/QA calls the Form-Completer once again to generate new values for the variables and the entire process begins again.

It should be noted that at the point when the student is asked the Tutor-generated question, he may in turn ask a question of his own (which activates Student/QA), work directly with the NLS file with a call to NLS, or quit for continuation at a later time.

The Form-Completer:

This specialist of the Tutor/QA system takes the incomplete semantic form and fills in values for the variables until the form is completed. Each type of semantic form has its own Form-Completer specialist. For example, the semantic form:

(QPARTS-IN-PART STATEMENT (BRANCH (ADDR XXX)))

requires that an address "XXX" of a branch in the NLS file be found. (The English interpretation of this form is the question, "WHAT STATEMENTS ARE IN BRANCH XXX?") A call is made to LISP-NLS to find a "good" branch, i.e. one that exists and that has at least one substatement. Other forms require calls to LISP-NLS to find good plexes, statement contents, levels of statements, etc. to use to fill out their semantic forms.

Some semantic forms require filling various cases. In a QFIND/INSTR semantic form (from "What command 'verbs' an 'object'"), the cases Agent, Verb, and Object must be filled. Since the question is directed to the student, the Agent case is filled automatically with "user". The Verb is randomly chosen from a list of verbs like move, copy, delete, print, etc. The selection of the Object is, of course, dependent on the Verb. To find an appropriate Object, the semantic network is queried. The chain of inferences that must be drawn for the verb "move" is as

follows: The Instrument for the verb "move" is retrieved, MOVE\COMMAND. Under MOVE\COMMAND is the Object on which it works, STRUCTURE\UNIT. Since its part of speech is a CN (concept noun) an example of it must be retrieved (XN). In the entry for STRUCTURE\UNIT are examples, STATEMENT, BRANCH, GROUP, and PLEX. The Object is chosen randomly from among these four, say BRANCH. The semantic form is now complete:

```
(QFIND/INSTR  ((AGENT USER)
               (VERB MOVE)
               (OBJ BRANCH)))
```

One last check is made to make sure that this semantic form has not been generated previously (to keep from asking the same question more than once). With this completed semantic form Tutor/QA simultaneously proceeds with the work of the De-Parser which derives the English surface representation of the request to present to the student, ("WHAT COMMAND MOVES A BRANCH?"), and the Evaluator which evaluates this form to derive the correct answer.

The Answer Comparer:

For each kind of semantic form there is an Answer Comparer specialist. At present, the semantic forms which Tutor/QA handles are such that answers to them are simple lists of items like "3A 3A1 3A2" as opposed to entire sentences (like the response to the question "How do you delete statement 3A1?" -- "I first type "delete" followed by

the word "statement", 3A1 and a carriage return. Then I type another carriage return after I see the OK: prompt.") Obviously the latter response from the student would be much more difficult to analyze, requiring a detailed parse and interpretation of the meaning of all the sentences involved, to say nothing of the detailed matching procedure that would be needed to see if that meaning was equivalent to the correct answer produced by the Evaluator.

Concerning ourselves with the former type of reply, the Answer Comparer looks at the match of the two responses, noting whether items are missing or extra in the student's reply. It then reports to the student appropriately and reports back to Tutor/QA how the student performed.

Future Considerations

The shared representation in Student/QA and Tutor/QA allows the addition of a very powerful mechanism, a history list, one list containing all Tutor-generated requests and the other all Student-generated requests, both in semantic form representation.

The first thing that falls out of having this feature is the ability in Student/QA mode to answer a student's procedural question and then to be able to respond to "DO IT" by picking up the top-most semantic form of a procedure request on the Student history list and executing it.

Although we now handle in a very limited way such "DO IT" requests, we have always assumed that such requests refer to performing the current task. Obviously this is inadequate.

Second, the history list feature provides us with the ability in Tutor/QA mode to recognize a "cheating" question by the student and to block it if we wish. For example, the Tutor asks:

WHAT STATEMENTS ARE AT LEVEL 2?

(a question produced from the semantic form

(QPARTS-IN-LEVEL (LEVEL 2))

Instead of responding, the student asks:

WHAT ARE THE STATEMENTS AT LEVEL 2?

This request is simply a paraphrase of the Tutor's question. We recognize this by comparing the parse (semantic form) for it with the form at the top of the Tutor's history list. (They will, of course, be the same.) We may then decide either to answer the question or to refuse to answer allowing him to ask other questions, but not one that parses into the same form as the Tutor's question.

The history list feature also gives the Tutor a simple repository for the questions it has asked -- a place to check on already-asked questions to keep from repeating itself.

SECTION IV - TASK EVALUATION

Task evaluation is potentially one of the most fertile areas of NLS Scholar, and at the same time is potentially one of the most overwhelming, due to its close connections with the nebulous areas of searching the space of discrepancies, learning from selected discrepancies, emulating the tutor's example and even simulating (crudely) a student's probable misunderstanding.

At present, task evaluation is limited to a comparison of the correct file, which it generates from the correct stored command sequence, with the student's file. It reports to the student the scope of his error by printing on his terminal the discrepant sections of his file and the corresponding sections of the correct file. Some sophistication is achieved by using "sensitive state" flags to limit the level of error description to terminology consistent with the student's current knowledge. In addition, there are specialist-reporters for file structure and content, CM position, viewspecs and printing which allow for special description of errors in these areas.

Sensitive states

Sensitive state flags affect how an error is reported. They are associated in the data base with each task. For example, the tasks in Lesson 1 have the flag CMLEVELGAG

associated with them because the student has not been told about branches and this prevents the CM specialist-reporter from pointing out same-branch relationships.

Other implemented sensitive state flags behave as follows. CMPLEXFLG enables the CM specialist-reporter to point out simple same-plex relationships. VSDESCRIBEFLG causes the viewspec specialist-reporter to talk about levels and lines, rather than x's, b's, etc. #RETRIES is really a counter flag that provides a limit on the length of time one can spend doing and redoing a task. The default is initially set to 2 trials.

Specialist-reporters

Four specialist-reporters have been implemented covering file structure and contents, CM position, viewspecs and printing. Our design strategy in each has been to classify and describe the extensional discrepancies between what was expected of the student and what the student actually did. In each case, some suitable range of error types and format for description was chosen to fit the particular aspect of the error. An analysis of each separate area yielded four independent formulae, with one exception: a generalized list-comparison algorithm was found to be applicable to exploring any two lists for insertions, omissions, and content errors, regardless of the form of

information ultimately to be extracted.

The specialist-reporter for file structure and content performs an analysis of the files into three cases: change extraneous, change omitted, and change incorrect or faulty. We have found it profitable to compare the student's file with the initial one, the target file with the initial, and then to compare the comparisons. The information extracted by this specialist is whichever section of the file is to be printed in order to show just the discrepant parts.

In doing viewspecs evaluation, a more detailed error-typing was possible. It was possible to add "overdone" and "underdone" classes (too much or too little printed). This not only produced output that was more to the point; it also permitted an appropriately selective task continuation criterion.

Retrial vs. Repair

Often it is more instructive to fix one's mistakes than to try again; but up until now, we have leaned towards retrial rather than repair. In general, this decision as to whether to stick with the present mistake is a difficult one. It involves having special knowledge about each command and about how much background and understanding can be presumed in the student.

Extensional vs. Intensional Information

In the present task evaluation system, only extensional information is used; that is, we look at the results of executing a sequence of commands, i.e. the NLS file itself, rather than looking directly at the sequence that produced it. Although this approach has proved quite effective, there is much power to be gained from analyzing the intensional information contained in the command sequence itself. This analysis would use knowledge from the data base to report to the student the consequence of an incorrect command sequence.

For example, if the correct command sequence requires the word "plex" and the student types "branch", the command sequence analyzer would report the error possibly using information from the data base to construct an explanation of the meaning of "branch" vs. "plex" and any other special information it deemed useful to review. This type of explanation provides a unique method of reviewing information about the use of NLS at points in the lesson where such review is obviously needed (at points where the student errs). data base of the meaning of "branch" vs. "plex".

SECTION V - SYSTEM ORGANIZATION

Overall Organization

The overall organization of NLS-SCHOLAR is represented in Figure 6. There is a system executive which controls and supervises the functioning of the four main modules of the system (DELIVERY, STUDENT/QA, TUTOR/QA, TASK MONITOR). The EXECUTIVE services these modules' requests and provides communication paths among them. When in tutorial mode (the normal mode in NLS-SCHOLAR), the EXECUTIVE is driven by the AGENDA which is a LISP representation of the Primer and is produced automatically from the Primer's content.

The DELIVERY module is very simple; it retrieves the string the EXECUTIVE wants to print to the student and prints it. If a question is asked of the student by the system, the ANSWER EVALUATOR is called to judge the correctness of the student's answer. TUTOR/QA can also call STUDENT/QA to allow the student to ask other questions rather than immediately answering the question posed to him.

The TASK MONITOR is called either by EXECUTIVE when a task must be set up for the student to perform, or by STUDENT/QA, when an NLS command must be performed to use the response in constructing an answer.

TASK MONITOR can perform commands in a number of

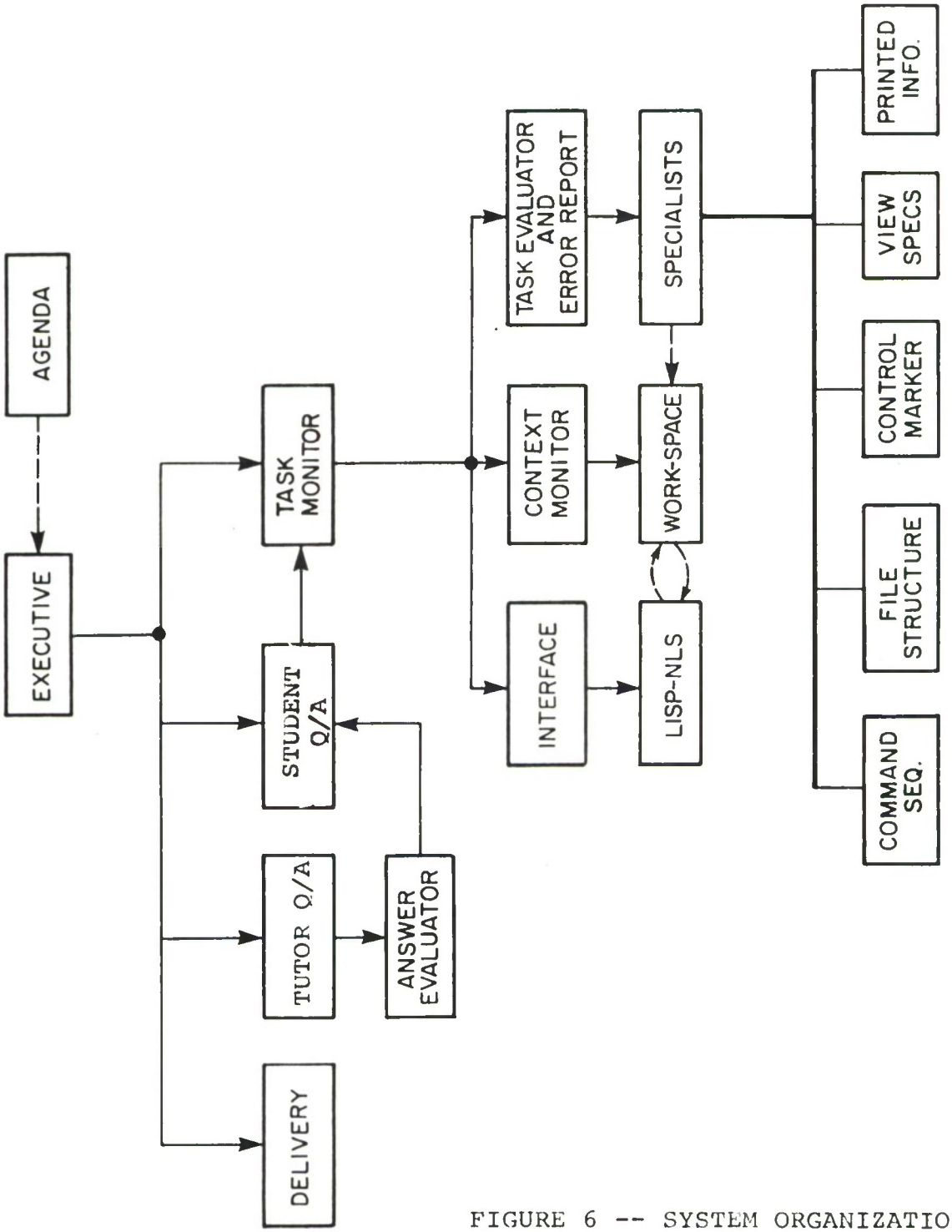

FIGURE 6 -- SYSTEM ORGANIZATION

different ways:

a) normally, by allowing the student to type on his terminal as in standard NLS use.

b) invisibly, by passing commands to LISP-NLS without any trace of their performance showing up in the student's terminal. The STUDENT/QA system often uses this mode as does the TASK MONITOR.)

c) in tutorial mode, imitating what a human tutor would do if he typed commands on the student's terminal. This is done by surrounding with parentheses what NLS-SCHOLAR "types" for the student.

The function that is responsible for all this, and the only one that has access to LISP-NLS is called INTERFACE.

When used in mode (a), TASK MONITOR uses CONTEXT MONITOR to make a copy of the initial state of the NLS file, and then puts the student in contact with LISP-NLS to allow him to type in his commands. When the student has finished, the state of his NLS file is saved for later use. Then TASK MONITOR obtains a target file, i.e. a representation of what the state of the NLS file would be if the student had performed the task correctly. TASK MONITOR does this by performing the correct command sequence for the task, which is stored in the data base. These commands are performed invisibly to the student, and they act on the copy of the student's file that was saved before. When this is done, the state of the file (TARG), the initial state of the file

(INIT), and what the student obtained with his commands (STU), are delivered to TASK EVALUATOR and ERROR REPORT.

Error Analysis

ERROR REPORT is responsible for analyzing the three file (INIT, STU, and TARG), deciding if significant errors have occurred, and if so, figuring out how to report these errors to the student. To do that, a function named TASK EVALUATOR is called. TASK EVALUATOR in turn can call several specialists to analyze the files and discover any errors that the student may have made in terms of the structure of the file, its content, the final position of the control marker, the state of the viewspecs, and whether or not he printed correctly whatever the task might have required.

An error in structure is always crucial and must be reported to the student. Therefore, an important function of ERROR REPORT is to provide him with a description of the error that is adequate for him to realize his mistake and "debug" his task commands. For example, the structure specialist operates by first checking if any differences exist between the student's file and the target file. If this is the case, then an environment of the error that is common to both files is determined. In order to frame the environment of the error, some statements immediately

preceding this environment in the initial file, and some statements following it, may be printed out to him.

How the system works

Let us now clarify this description with an example. Consider the protocol presented in Section I. EXECUTIVE retrieves from the AGENDA its "instructions", which in this case consists of delivering the text headed by "SUBSTITUTE WORD IN STATEMENT", performing a task as if the tutor were demonstrating how to do it, delivering some more text, and finally giving the student a task to perform.

When NLS-SCHOLAR demonstrates to the student how to perform a command, EXECUTIVE calls TASK MONITOR, hands it the correct command sequence for the task, and instructs it to print out, using the parenthesis notation that we have adopted to show the student what the "tutor" is actually typing. TASK MONITOR then sets up the appropriate call to INTERFACE, and LISP-NLS performs the commands. Since the task in this case is guaranteed to be correct, there is no need to let ERROR REPORT intervene, and TASK MONITOR returns to the EXECUTIVE. After more text has been delivered, and TASK 41 completed, the EXECUTIVE calls STUDENT/QA to handle student questions.

Consider, for example, the question the student asks:

What statement contains "CORN"?

Here, TASK MONITOR returns to the EXECUTIVE which then calls STUDENT/QA. This question concerns the state of the NLS file and cannot be answered with information stored in the semantic network. Therefore, STUDENT/QA constructs a command for LISP-NLS to execute and uses the results in constructing an answer. In our case, STUDENT/QA calls TASK MONITOR and asks it to perform the commands JUMP Ø and then JUMP "CORN" and \ (back slash), which in NLS cause the control marker to jump to the statement containing "CORN" and print its address and content. The command is performed invisibly and the response is returned to STUDENT/QA which then extracts the address and constructs the answer. The context manipulation machinery meanwhile took care of protecting the student´s environment by providing a scratch copy of it on which these commands were performed.

At this point the EXECUTIVE again turns to the AGENDA to find out what to do next. In this case, the AGENDA requires delivering more text ("Use the Substitute command to change statement 2A1..."). Let´s see what happens when the student performs this substitution task. TASK MONITOR is called, and it orders INTERFACE to let the student talk directly to LISP-NLS. After he types "Done", TASK MONITOR saves the student´s environent and sets up the NLS file to its initial state by calling CONTEXT MONITOR.

In this way, TASK MONITOR can now use the preferred

command sequence to find out what the NLS file should look like if the task were performed correctly. TASK MONITOR does that by performing invisibly (via the appropriate call to INTERFACE) the preferred command sequence on the initial file, thus obtaining the target file. With the student, initial, and target files now obtained, TASK MONITOR calls ERROR REPORT and TASK EVALUATOR. The structure specialist detects a difference in branch 2A and returns to ERROR REPORT, which figures out how to tell the student what happened. ERROR REPORT does that by synthesizing a command to print branch 2A of the target file, and this command is performed, without echoes being shown, via a return to TASK MONITOR and a call to INTERFACE. After that, control returns to ERROR REPORT which synthesizes another print command to describe what the student did instead, and the same sequence is repeated. ("But you changed this part of it...") this time using the student's file. After all this is done, TASK MONITOR asks CONTEXT MONITOR to restore things to their initial state and the student is requested to perform the task again. The structure specialist detects a difference in branch 2A and returns to ERROR REPORT, which figures out how to tell the student what happened. ERROR REPORT does that by synthesizing a command to print branch 2A of the target file, and this command is performed, without echoes being shown, via a return to TASK MONITOR and a call to INTERFACE. After that, control returns to ERROR

REPORT which synthesizes another print command to describe what the student did instead, and the same sequence is repeated. ("But you changed this part of it...") this time using the student's file. After all this is done, TASK MONITOR asks CONTEXT MONITOR to restore things to their initial state and the student is requested to perform the task again.

Student Aids

Several facilities have been developed to facilitate the use of NLS-SCHOLAR in tutorial mode. The CONTROL module allows the stdent to type *NLS* to the Question Answering system at anytime, and gain access to NLS for free play and interaction, without disturbing the state of his file and therefore not altering the progress of the lesson. The student may type *RESTART* to restart performing a task, with all the commands performed so far being forgotten. He can type *QUIT* to quit a lesson at anytime, without waiting for the end of it; he may type *PROCEED* to continue it again.

Debugging Aids

NLS-SCHOLAR contains a DRIBBLE facility to aid in the debugging of the system. Whenever someone uses NLS-SCHOLAR, a complete record of the transaction is kept on a protocol

file. Also, whenever a sensitive portion of the program fails, a message (via SNDMSG) is sent to the person who wrote that part of the program, and relevant information about the failure is written in a special file in the programmer's directory so that he may examine the problem and correct it.

NLS-SCHOLAR can be run in human-backed mode, when special arrangements have been made. This mode allows a human expert watch (via linked terminals) the student/computer dialog, and step in when the system fails. For example, if the Question Answering system fails to understand a question by the student, the human expert can provide the answer by typing it in his own terminal.

SECTION VI - CONCLUSIONS AND RECOMMENDATIONS

The "finality" of the present report is only an administrative technicality; much remains to be done before NLS-SCHOLAR can be considered finished and ready to use as a stand-alone Help and Tutorial facility. However, we have made good progress towards that end, and we feel that even now, in spite of the systems's limitations, it could be useful to its users. We believe that NLS-SCHOLAR offers some very positive advantages that could make it worthwhile to its users, even in its present, unfinished state: the lessons are very nicely organized, and the systems's ability to present examples, show how to do things, and propose tasks which it then evaluates and comments upon, is very powerful.

To make the system operational requires work in the following aspects:

1) A heterarchical, rather than hierarchical control structure is needed to allow the student more freedom and flexibility
2) Expanding the tutorial facilities to cover the most commonly used and useful commands in the NLS editing system.
3) Inproving the way mistakes are pointed out, by showing the student what's wrong with his solution rather than point out what's wrong with his result.

4) Improving the system's human engineering aspects and efficiency of operation, i.e. enabling students to proceed with the lessons at their own pace, quitting and resuming a lesson whenever they want.

We expect that in the next phase of our work, having upgraded the system along the lines described above, we will be able to come to grips with the real problems that a system of this sort encounters in an operational environment.

REFERENCES

[1] "TNLS Users' Guide" November 1973, obtainable from Augmentation Research Institute, Menlo Park, Calif. 94025.

[2] Carbonell, Jaime R. "AI in CAI: An Artificial-Intelligence Approach to Computer Assisted Instruction" IEEE Transactions on Man-Machine Systems, MMS-11, New York, December 1970.

[3] Carbonell, Jaime R. and Collins, Allan M. "Mixed-Initiative Systems for Training and Decision-Aid Applications" ESD-TR-70-373, November 1970.

[4] Collins, Allan M., Warnock, E.H. and Passafiume, J.J. "Analysis and Synthesis of Tutorial Dialogues" in Advances in Learning and Motivation Vol. 9, G.H. Bower, Ed., Academic Press, in press.

[5] Grignetti, Mario C. and Warnock, Eleanor H. "Mixed-Initiative Information System for Computer-Aided Training and Decision-Making" ESD-TR-73-290, September 1973.

[6] Brown, John Seely, Burton, R.R., and Bell, A.G. "SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI)" BBN Report No. 2790, March 1974.

[7] Brown, John Seely and Burton, Richard R. "SOPHIE: A Pragmatic Use of Artificial Intelligence in CAI" Proceedings of the National ACM Conference, San Diego, California, November 1974.

[8] Watson, Richard W. "National Software Work Developments - A Technical Proposal". SRI-ARC Journal #23352, July 1974.

[9] Carbonell, J.R. and Collins, A.M. "Natural Semantics in Artificial Intelligence" in Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, 1973. Reprinted in the American Journal of Computational Linguistics, 1, Mfc 3, 1974.

[10] Teitelman, Warren; et al. "INTERLISP Reference Manual", 1974.

[11] Collins, A.M., Passafiume, J.J., Gould, L., Carbonell, J.G., "Improving Interactive Capabilities in Computer-Assisted-Instruction," Cambridge Massachusetts: Bolt, Beranek and Newman, BBN Report No. 2631, 1973.

APPENDIX

Complete Scenario (Primer)

This Appendix is meant to fulfill two roles:

a) a complete version of the didactic material available at present under NLS-SCHOLAR, and

b) a more complete demonstration protocol than the one presented in Section I.

In this regard, the reader will find far more descriptive user-system interactions, especially with respect to the Print and Viewspec Specialist-Reporters (see TASK 54 and following).

LESSON ONE

INTRODUCTION

Hello. Welcome to your first lesson about NLS - the 'oN Line System' developed by Douglas Engelbart and his staff at Stanford Research Institute.

I'll be describing some parts of this system to you, showing you how to use it, and giving you tasks to perform.

From time to time I'll stop and ask if you have any questions by printing 'QUESTIONS?' in the margin, followed by a '*' on the line below. If you have no questions at that point, just type another '*' followed by a <CR> and I'll continue. If you do have a question, type it in after the '*' and terminate it with a '*' and a <CR>. I'll try to answer it and then I'll print another '*' so you can ask another question if you wish. When you have no more questions, type '* <CR>' and I'll continue.

QUESTIONS?

What does <CR> stand for?

<CR> IDENTIFIES THE RETURN KEY ON A KEYBOARD.

**

NLS is a text-processing system which is used as an aid in writing, re-organizing, indexing, publishing, and disseminating information of all kinds.

NLS is a very large system made up of many subsystems. In what follows I will introduce only some of the file-handling, text-editing, and simple printing commands which are used in the EDIT subsystem.

You will be presented with a series of tasks to perform using the EDIT subsystem of NLS. Everytime I propose a task for you to do, I'll put you in touch with NLS so it can execute your commands. When I do this, the characters EDIT C: will appear in the left margin, like this:

EDIT C:

The characters EDIT C: are printed by NLS. The EDIT part is a 'herald' indicating that you are talking to the main subsystem of NLS, the one called EDIT. The C: part is a 'prompt' indicating that the EDIT subsystem is waiting for you to type a command.

After you finish typing in your command, NLS will perform whatever action you requested. When this action has been completed, NLS will print another EDIT C: in the margin, indicating that it is awaiting your next command. When your task is completed -- when you don't wish to give any more commands -- type 'Done <CR>' to tell NLS that you wish me to continue.

If you make a mistake while typing a command you can repair it by typing <CTRL-A>, or you can start over by typing <CTRL-X>. Typing <CTRL-A> will erase the last character you typed. It's kind of like

the backspace on a typewriter, and you can use it successively twice to erase the last two characters, three times to erase the last three, etc.

Typing <CTRL-X> will cancel the entire command and will allow you to start over.

QUESTIONS?

What are the file handling commands?

EXAMPLES OF A FILE-HANDLING COMMAND ARE THE LOAD COMMAND, THE UPDATE COMMAND, AND THE CREATE COMMAND.

Give me some examples of printing commands.

EXAMPLES OF A PRINTING COMMAND ARE THE PRINT COMMAND, THE <LF> COMMAND, THE ^ COMMAND, AND THE \ COMMAND.

What does the <LF> command do?

THE <LF> COMMAND PRINTS THE NEXT STATEMENT.

What is a statement?

A STATEMENT IS THE BASIC STRUCTURE UNIT OF AN NLS FILE.

What is a prompt?

A PROMPT IS A SIGNAL FROM NLS THAT IT IS AWAITING AN INPUT STRING FROM THE USER.

How do I complete a task?

THE USER COMPLETES A TASK USING THE DONE COMMAND,
PROCEDURE: HE TYPES 'DONE', FOLLOWED BY A <SPACE>, AND <CR>.

How does one cancel a command?

THE USER CANCELS THE CURRENT NLS COMMAND USING THE CTRL-X COMMAND.

**

NLS FILES

In order to begin using NLS you will need to specify which 'file' of information you want to work with. Each file is sort of like a notebook or folder in which you can keep information; you may keep a

different notebook (file) for each separate subject in which you are interested.

Files are automatically stored when you are not using them. Before you can work with a file you must 'load' it from storage. Each file has a name so you may refer to it easily.

I'd like to show you the file named DINNER so you can see how an NLS file is structured. Your first task is to load this file so you can work with it. When the EDIT C: appears, type the command:

Load File DINNER <CR>

As this single command completes the task, type:

Done <CR>

when the next EDIT C: appears.

Note that you should terminate each part of the command with a space; you should terminate the entire command with a carriage return. Please be sure to type a space between 'Done' and the <CR>.

The characters that you type on your terminal will be interspersed with some 'prompts' that NLS will print between your words. Please don't let this disturb you.

TASK 1

EDIT C: Load C: File T: DINNER ;
(NLS-SCHOLAR, DINNER.LNLS;3,)
EDIT C: Done OK:_;

NLS prints a C: prompt whenever it expects a command term. Most NLS commands, like the Load command, are composed of two command terms, usually a verb followed by a noun.

NLS prints a T: prompt whenever it expects you to type some text, in this case the name of the file to be loaded.

The file DINNER is now the current file with which you are working. Any editing commands which you use will automatically apply to that file until a different file is loaded.

QUESTIONS?

What does the C: mean?

THE C: PROMPT IS A PROMPT THAT INDICATES
THAT NLS IS WAITING FOR A COMMAND TERM FROM
THE USER.

What is the T: prompt for?

THE T: PROMPT IS A PROMPT THAT INDICATES
THAT NLS IS WAITING FOR SOME TEXT FROM THE USER.

**

PRINTING A FILE

Your next task is to use the Print command to see the content of the DINNER file. You do that by typing 'Print <CR>'. (Again, please don't be disturbed by the prompts.) After the entire file has been printed, don't forget to type 'Done <CR>' when the EDIT herald appears again.

TASK 2

```
EDIT C: Print OK:/C: _;
  <NLS-SCHOLAR>DINNER.LNLS;3,  28-AUG-74 08:28 CLH ;
    1 SOUP
       1A TOMATO
       1B VEGETABLE
       1C CREAM OF MUSHROOM
    2 ENTREE
       2A FRIED CHICKEN
       2B PRIME RIBS
       2C SCALLOPS
          2C1 BROILED
          2C2 FRIED
       2D SALMON
          2D1 WITH CREAM SAUCE
    3 DESSERT
       3A PIE
          3A1 RHUBARB
          3A2 BLUEBERRY
       3B STRAWBERRY SHORTCAKE
       3C ICE CREAM
          3C1 PEPPERMINT
          3C2 MAPLENUT
          3C3 CHOCOLATE
EDIT C: Done OK:_;
```

Please tear the paper off here and place it on the table beside you so you can refer to this file easily in the future.

After 'Print' you could type various nouns indicating that only part of the file is to be printed. But if no noun is supplied, then the single command term 'Print', followed by a <CR>, will print the entire content of a newly-loaded file.

The mysterious prompt OK:/C: which NLS printed above indicates that it expects either confirmation (OK:), which you provided by typing <CR>, or another command term (C:). We shall return to the nouns that you can type after 'Print' later.

QUESTIONS?

What can I type after Print?

THE THINGS THAT CAN FOLLOW PRINT ARE:
Branch Statement Plex Group

How do I print a statement?

THE USER PRINTS A STATEMENT USING THE PRINT COMMAND,
PROCEDURE: HE TYPES 'PRINT ', FOLLOWED BY
'STATEMENT ', AN ADDRESS, <CR>, VIEWSPECS, AND
<CR>.

**

OUTLINE STRUCTURE

Let's look now at the information in the file. Notice that there is a heading at the top of your file which consists of some identifying information. The first part within the angle brackets, NLS-SCHOLAR, is the 'directory name'. Directories are used to group together and index file names. The second part, DINNER, is the 'file name'. The third part, LNLS, an extension of the file name. The fourth part is a 'version number'. Thus the heading tells you that this is the first version that's been made of the file named DINNER.LNLS which is in the NLS-SCHOLAR directory.

The rest of the heading consists of the date and the time of the file's creation, and the initials of the person who created it.

This information is supplied by NLS as the content of a special statement called 'statement zero'. It is the only statement whose initial content is supplied by NLS instead of by the user.

Following statement zero, you can see that an NLS file is structured like a standard indented outline. The numbers on the left are called 'statement numbers'. Statement zero is the only one whose statement number is not visible. The text following each statement number is called a 'statement'. The statements in this example are very short; however, statements may contain up to 2000 characters each and often consist of an entire paragraph of text.

The statement is the basic 'structure unit' of an NLS file. Larger structure units called 'branches', 'groups', and 'plexes' are formed from sets of statements. Of these, only branches will be of interest to us initially.

The DINNER file has three main branches: branch 1, branch 2, and branch 3. Branch 1 consists of all statements whose statement numbers begin with 1; branch 2 of all statements whose numbers begin with 2; branch 3 similarly. The file contains several smaller branches as well.

WHAT STATEMENTS ARE IN BRANCH 3A? (Please put spaces between the statement numbers and terminate your list with a '*' followed by a <CR>.)

3A1 and 3A2

CORRECT. HOWEVER, YOUR ANSWER IS INCOMPLETE.
3A1 AND 3A2 ARE O.K.
YOU COULD ALSO INCLUDE 3A.

WHAT STATEMENTS ARE IN BRANCH 1?
1, 1A, 1B, and 1C.

VERY GOOD.

Individual statements may be inserted, deleted, moved, transposed, and copied within the file. Larger structure units like branches may be similarly manipulated.

Within a statement the basic 'string units' are 'words' and 'characters'. A word has its usual intuitive definition; a character is any individual symbol (letter, number, punctuation mark, space, etc.) you can type on your keyboard. Individual words and characters may also be inserted, deleted, changed, etc.

The terms statement, branch, plex, group, word, and character are important because they are the nouns used as the second term of many NLS commands.

Statement numbers are used to indicate the relative position of each statement within the file. They are one kind of 'address' by which you may specify or 'point to' a particular statement. Statement numbers are assigned by the NLS system and may be changed by NLS when new statements are inserted or old ones deleted.

QUESTIONS?

What statements are in branch 3A?

THE STATEMENTS IN BRANCH 3A ARE:
3A 3A1 3A2

Give me a definition of statement zero.

STATEMENT 0 IS A STATEMENT AT THE BEGINNING
OF EVERY NLS FILE CONTAINING IDENTIFYING
INFORMATION.

What is the content of statement zero now?

THE STATEMENT 0 IS:
<NLS-SCHOLAR>DINNER.LNLS;3, 28-AUG-74 08:28 CLH ;

**

DELETING A STATEMENT

To see how statement numbers are changed by NLS, let's pretend that we've run out of tomato soup, rhubarb pie, and strawberry shortcake and wish to delete these items from our menu. I'll delete the statement containing "TOMATO", using the Delete command.

```
EDIT C: (Delete ) C: (Statement ) at A: (1A)(<CR>);
 OK: (<CR>);
EDIT C: (Done ) OK: (<CR>);
```

I typed:

```
Delete Statement 1A <CR>
```

to produce the first line; the remaining characters were printed by NLS. Whenever I type something, I'll indicate it by enclosing it within parentheses.

You're already familiar with the C: prompt for command term. The A: prompt is for an address; here I provided the statement number of the statement to be deleted. Notice that I typed a <CR> after this address to terminate the command. If I had typed an address which did not exist in the file, such as 1D3, NLS would have printed a '?'. Then it would have given me another A: prompt so I could supply a valid address instead.

NLS asks OK: because it wants to confirm that I really mean to delete statement 1A. (Once a statement has been deleted, it may be impossible to recover its content.) I gave confirmation by typing <CR>; if it wasn't OK I would have cancelled the command by typing <CTRL-X>.

QUESTIONS?

**

Your next task is to delete the statement containing "RHUBARB". Before you start on that I'd like to remind you that there's another place at which you can ask questions -- wherever there's an EDIT herald in the margin. Instead of giving a command term you can type an asterisk indicating that you wish to ask me a question. Then type in your question, terminating it as usual with another asterisk. I will attempt to answer your question and will then return you to NLS which will print another EDIT herald in the margin. At that point, you can either ask another question by typing an asterisk, or you can type a command term. Thus if you don't know how to perform a given task, you can ask questions about it.

Please use the Delete command to delete the statement containing "RHUBARB" from the menu.

TASK 3

EDIT C: *What is the address of the statement containing "RHUBARB"?*

THE ADDRESS OF THE STATEMENT CONTAINING "RHUBARB" IS:
 3A1

continuation of TASK 3

EDIT C: *Where is the "RHUBARB"?*

THE ADDRESS OF "RHUBARB" IS:
3A1 + 7

continuation of TASK 3

EDIT C: *Where is the "TOMATO"?*

I CAN'T FIND AN ADDRESS FOR THAT.

continuation of TASK 3

EDIT C: Delete C: Statement at A: 3A1;
 OK:_;
EDIT C: Done OK:_;

Very good. Now delete the statement containing "STRAWBERRY SHORTCAKE".

TASK 4

EDIT C: Delete C: Statement at A: 3B;
 OK:_;
EDIT C: Done OK:_;

Fine. Now the statements containing "TOMATO", "RHUBARB", AND "STRAWBERRY SHORTCAKE" have all been deleted from the DINNER file.

QUESTIONS?

**

DELETING A BRANCH

Now let's suppose we've run out of scallops as well. Watch me try to delete the statement containing "SCALLOPS".

```
EDIT C: (Delete ) C: (Statement ) at A: (2C)(<CR>);
 OK: (<CR>);
ILLEGAL DELETE
EDIT C: (Done ) OK: (<CR>);
```

NLS refused to delete statement 2C because it has the two ´substatements´ 2C1 and 2C2. It is not possible to delete any statement having substatements.

In order to remove the statement "SCALLOPS" from our menu we will have to delete the whole branch 2C, not just the statement 2C. (This is reasonable since "BROILED" and "FRIED" apply only to the statement "SCALLOPS" anyway.) Do that now, using the Delete command with ´Branch´ as the noun instead of ´Statement´.

TASK 6

```
EDIT C: Delete C: Branch at A: 2C;
 OK:_;
EDIT C: Done OK:_;
```

NLS will also refuse to delete (or to move) statement zero or branch zero since the information in statement zero is needed at the head of the file.

QUESTIONS?

**

THE RENUMBERED FILE

Now let me show you what happens when I try to delete the statement containing "BLUEBERRY" which used to follow the statement containing "RHUBARB" on the menu.

```
EDIT C: (Delete ) C: (Statement ) at A: (3A2)(<CR>);
 ?   A: (<CTRL-X>) ##
EDIT C: (Done ) OK: (<CR>);
```

NLS prints ´?´ because the DINNER file no longer contains a statement with address 3A2. I typed <CTRL-X> after the A: to cancel the command instead of providing a new address. (Note that <CTRL-X> echoes as ##.)

When the statement containing "RHUBARB", 3A1, was deleted, the statement containing "BLUEBERRY" which was formerly 3A2 was ´promoted´ to become statement 3A1. So to delete the statement containing "BLUEBERRY" we would have to delete statement 3A1 again.

Because some statements may be renumbered in this way whenever a deletion occurs, it is important to update your view of the file.

Let me print the modified DINNER file for you now so you can see that

all our deletions have indeed occurred and what effect they have had on the numbering of the file.

```
EDIT C: (Jump ) to A: (0)(<CR>);
EDIT C: (Print ) OK:/C: (<CR>);
  <NLS-SCHOLAR>DINNER.LNLS;3,  28-AUG-74 08:28 CLH ;
    1 SOUP
       1A VEGETABLE
       1B CREAM OF MUSHROOM
    2 ENTREE
       2A FRIED CHICKEN
       2B PRIME RIBS
       2C SALMON
          2C1 WITH CREAM SAUCE
    3 DESSERT
       3A PIE
          3A1 BLUEBERRY
       3B ICE CREAM
          3B1 PEPPERMINT
          3B2 MAPLENUT
          3B3 CHOCOLATE
EDIT C: (Done ) OK: (<CR>);
```

Please tear off the paper again to compare this file with its earlier version.

Note how the statement numbers have been changed by NLS. You can see that many statements have been renumbered (´promoted´), some of them acquiring the statement numbers of the deleted statements. Although statements 1A, 2C, 2C1, 3A1, and 3B were all deleted, these statement numbers still exist in our file -- but the statements contents are now different.

QUESTIONS?

What is the purpose of that jump command that you typed?

THE JUMP COMMAND MOVES THE CM.

What is the CM?

THE CM IS THE CONTROL MARKER THAT POINTS AT THE CURRENT CHARACTER.

**

THE CONTROL MARKER (CM)

You may have noticed that instead of just typing ´Print <CR>´ as before to print the DINNER file, I typed ´Jump 0 <CR>´ first. The reason for

this is as follows.

Whenever you are working with a file there is always a pointer called the 'control marker' (CM) which points directly at some character in the file. It indicates the current address within the file -- the place where you are currently working.

Most editing commands cause the CM to be moved. For example, when a file is first loaded the CM points to the first character of the file. When you insert or delete some statements, the CM is moved.

The Print command followed by <CR> actually prints the content of the current file from the current position of the CM all the way to the end. Thus when a file is newly-loaded, 'Print <CR>' causes all of it to be printed. The Print command does not change the position of the CM.

QUESTIONS?

**

CM COMMANDS

The Jump command is used to move the CM anywhere within the current file. I moved the CM to the beginning of statement 0 so that the whole file would be printed.

Use the Jump command yourself now to move the CM to the statement containing "BLUEBERRY". (Please be sure to consult the latest version of the DINNER file as the statement numbers have been changed.)

TASK 7

```
EDIT C:  Jump to A:  3A1;
EDIT C:  Done OK:_ ;
```

To see the effect of this command, type just a single period, followed by a space, after the EDIT herald.

TASK 8

```
EDIT C:  . =3A1 +1
EDIT C:  Done OK:_ ;
```

This one-character command may be used following any EDIT herald to find the current position of the CM. The first number after the equals sign is the statement number; the second number after the plus sign is the character position within the statement to which the CM is pointing. Thus the CM is now positioned at the first character of statement 3A1.

QUESTIONS?

**

By using the Jump command in conjunction with the Print command you can start printing the current file wherever you like. Please use these commands now to print just the three ice cream flavors on your menu.

TASK 9

```
EDIT C:  Jump to A:  3B1;
EDIT C:  Print OK:/C:__;
           3B1  PEPPERMINT
           3B2  MAPLENUT
           3B3  CHOCOLATE
EDIT C:  Done OK:__;
```

Now use the '. ' command again to see what has happened to the CM.

TASK 10

```
EDIT C:  .__ =3B1 +1
EDIT C:  Done OK:__;
```

This shows you that 'Print <CR>' does not cause the CM to be moved. The current position of the CM is still 3B +1 where you moved it with the Jump command.

QUESTIONS?

**

CONTENT ADDRESSING

The address of a statement may be specified in a number of different ways. One thing you may type after an A: prompt, as you've seen, is a statement number.

Another thing you may type is a <CR>, which means that the address you want is the current address.

A third thing you may do is to specify the statement you want in terms of its content rather than its statement number. For example, to jump to the statement "BLUEBERRY", instead of typing:

Jump 3A1 <CR>

you could type:

Jump "BLUEBERRY" <CR>

with virtually the same effect. Using a content string as an address relieves you from having to keep track of the current statement numbers.

Try content addressing now by jumping to the statement "CHOCOLATE". Remember that your content string must be enclosed in double quotes and followed by a <CR>.

TASK 11

```
EDIT C:  Jump to A:  "CHOCOLATE";
EDIT C:  Done OK:__;
```

You can determine the statement number of this statement by using the . command again. Remember that this command is terminated by a space, rather than a <CR>. Try that now.

TASK 12

```
EDIT C:  . =3B3 +9
EDIT C:  Done OK: ;
```

Note that the CM is pointing to the ninth character of statement 3B3, not to its first character. When content addressing is used, the CM always points to that character of the statement which matches the last character of the content string.

QUESTIONS?

```
**
```

Now see what happens when I try to use content addressing to jump to the statement "FRIED CHICKEN".

```
EDIT C:  (Jump ) to A:  ("FRIED CHICKEN")(<CR>);
 ?  A:  (<CTRL-X>)##
EDIT C:  (Done ) OK:  (<CR>);
```

Here NLS is telling you that it can't find a statement containing "FRIED CHICKEN" and it asks for another address. (I cancelled this command by typing a <CTRL-X> after the second A: prompt.)

This is puzzling since "FRIED CHICKEN" is the content of statement 2A. NLS can't find this statement because it always starts searching at the current address (in this case 3B3 +9) and stops searching when it reaches the end of the file.

When using content addressing, you must always be sure that the CM is pointing to a statement which is earlier in the file than the one you want to find. One way to be sure is to adopt the habit of always moving the CM to statement zero before using a content address. Try to jump to the statement "FRIED CHICKEN" again, by jumping to statement zero first.

TASK 13

```
EDIT C:  Jump to A:  0;
EDIT C:  Jump to A:  "FRIED CHICKEN";
EDIT C:  Done OK: ;
```

Good. Now print the current position of the CM to determine where you are.

TASK 14

```
EDIT C:  . =2A +13
EDIT C:  Done OK: ;
```

When using content addressing it is not necessary to type the entire content of the statement or even an entire word. Any string of

consecutive characters which uniquely identifies the statement is sufficient. The file is searched sequentially and the first statement which contains the content string is selected. If the current position of the CM is statement zero, statement zero is searched first, then all statements of branch 1, followed by all statements of branch 2, etc.

QUESTIONS?

**

ONE-CHARACTER COMMANDS THAT PRINT

The statement at which the CM is pointing is called the 'current statement'. You can print the current statement using the \ command. The procedure is to type '\' followed by a space. Try that now and see what happens.

TASK 15

```
EDIT C: \
 2A  FRIED CHICKEN
EDIT C:  Done OK: _;
```

The statement directly before the current statement is said to be 'back' from it. To print the back statement you can use the ↑ command. The procedure is to type the single character '↑' followed by a space. Try it.

TASK 16

```
EDIT C: ↑
 2  ENTREE
EDIT C:  Done OK: _;
```

The ↑ command moves the CM. Use the . command to find the current position of the CM.

TASK 16A

```
EDIT C:  . =2 +1
EDIT C:  Done OK: _;
```

As you can see, the current statement is now statement 2.

The statement directly after the current statement is said to be 'next' to it. To print the next statement, you can use the <LF> command. The procedure is to type the key which says 'LINE FEED', followed by a space. (The LINE FEED key will be denoted as <LF>; it will echo as '%'.) Do that now.

TASK 17

```
EDIT C: %

 2A  FRIED CHICKEN
EDIT C:  Done OK: _;
```

As you may have expected, the <LF> command also moves the CM. What do you think the current statement is now? (Be sure to terminate your answer with '* <CR>'.)

2A

THAT'S RIGHT.

QUESTIONS?

**

UPDATING YOUR FILE

That's about enough for the first lesson. Before quitting, however, you should 'update' your file so that the changes you and I have made during this session will be incorporated into the DINNER file. At present these changes are stored in what is called its 'modification file', rather than in the main file itself. Type:

Update File <CR>

and all the information in the modification file will become part of the main file.

TASK 18

EDIT C: Update C: File OK:/C:_;
EDIT C: Done OK:_;

Good. The modification file is now empty, ready to receive new changes. There is no need to specify that the main file is to be stored away for further use. NLS does that automatically for you whenever you load a different file to work with, or when you leave NLS (or logout).

QUESTIONS?

**

REVIEW OF LESSON ONE

Lesson One has introduced the following commands:

Load command
Print command
Delete command for statements and branches
Jump command
Update command
\ (print the current statement)
↑ (print the back statement)
<LF> (print the next statement)
. (print the current position of the CM)

If you are unsure about any of these commands, this would be a good time to ask questions about them.

QUESTIONS?

**

Lesson One has introduced the following concepts:

structure unit
string unit
statement
branch
word
character

NLS command
one-character command
command term
control marker (CM)
<CR>
<CTRL-X>
<LF>

herald
prompt
  C:
  T:
  A:
  OK:
  OK:/C:

current statement
next statement
back statement
statement zero
substatement

file
file name
current file
modification file
outline structure

file specification
directory
extension
version number

address
statement number
content addressing
current address
current position of the CM

Again, if you have questions about any of these concepts, please ask them now.

QUESTIONS?

**

If you would like to practice what you have learned you may now print or modify any part of the DINNER file that you wish. The modification file that you create by doing that will not be kept, so your file will not be permanently changed. However, you must not use the Update command or your file may not be in a suitable form for doing Lesson Two.

When you don't wish to give any more commands, or ask any more questions, type 'Done <CR>' after the EDIT herald.

TASK 19

EDIT C: Done OK:_;

Lesson Two will teach you more about the structure of NLS files. It will teach you how to insert statements, how to create a new file, and how to modify existing statements without deleting them. Au revoir.

LESSON 2

ECHOING

Hello. Nice to meet you again.

I'd like you to start by loading your DINNER file again and printing it so we can see how to insert some more statements into it.

Before you do that, though, I'd like to confess that in Lesson One both you and I did more typing than was really necessary to indicate which commands we wanted. To specify each command term you need to type only enough characters to identify that term uniquely. As soon as enough characters are typed, you may hit the space bar and NLS will 'echo' the rest of the characters, just as if you had typed the entire command yourself. If you haven't provided enough characters, NLS will ring a bell indicating that it needs more and will wait for you to provide them. (Three characters are always sufficient.) Similarly if you type a character which cannot possibly result in a valid command term, a bell will ring and the character will not be accepted.

To see how this works, use the Load command to load the DINNER file. Use as few characters in 'Load' and in 'File' as is possible for recognition. (Please remember to type 'Done <CR>' afterwards.)

TASK 20

```
EDIT C:  Load C:  File T:  DINNER;
( NLS-SCHOLAR, DINNER.LNLS;2, )
EDIT C:  Done OK:_;
```

Please print this file so you can see its content. Again, practice using only a few characters to specify 'Print'.

TASK 21

```
EDIT C:  Print OK:/C:_;
  <NLS-SCHOLAR>DINNER.LNLS;2  1-OCT-74  08:28  CLH  ;
    1  SOUP
       1A  VEGETABLE
       1B  CREAM OF MUSHROOM
    2  ENTREE
       2A  FRIED CHICKEN
       2B  PRIME RIBS
       2C  SALMON
           2C1  WITH CREAM SAUCE
    3  DESSERT
       3A  PIE
           3A1  BLUEBERRY
       3B  ICE CREAM
           3B1  PEPPERMINT
           3B2  MAPLENUT
           3B3  CHOCOLATE
EDIT C:  Done OK:_;
```

Please tear off the paper here so you can refer to this file easily.

Note that the information in statement zero has been changed to indicate that this is version 2 of the file rather than version 1. Every time a file is updated, its version number is incremented by 1.

QUESTIONS?

**

LEVELS AND FIELDS

Let's look more closely at the structure of this file, by examining the composition of the statement numbers.

Statement numbers are composed of alternating 'fields' of digits and letters. The number of fields specifies the 'level' of the statement within the file. For example, statement 1A is at level 2.

What is the level of statement 3B3?
(Please remember to terminate your answer with '* <CR>'.)

3

THAT'S RIGHT.

What statements are at level 1?

1 2 3

VERY GOOD.

Although the statement number 0 consists of only one field, statement zero is not at level 1. Statement zero is an exception to the general rule. It is at level zero, the highest level in the file. Since all statements are at a lower level than statement zero, branch zero consists of all the statements in the file.

QUESTIONS?

**

Level numbers are never used in specifying NLS commands. However it is important to understand what they mean. When you insert a new statement in a file you must specify which statement it is to follow and also any difference in level between the new statement and the one before it.

Differences in level are described with the terms 'up' and 'down'. For example, if you wished to insert a statement to be numbered 1A, you would specify that it was to follow statement 1 one level down. If you wished to insert a statement 2 you could specify that it was to follow statement 1 at the same level.

If a branch 1 already existed which looked as follows:

```
1
   1A
   1B
      1B1
```

It would also be possible to insert statement 2 by saying that it followed statement 1B one level up, or that it followed statement 1B1 two levels up.

All insertions may be specified using only the terms 'down' and 'same'; however 'up' is provided as an added convenience.

QUESTIONS?

**

INSERTING A STATEMENT

To insert a new statement in a file, one uses the Insert command and gives the address of the statement after which the insertion is to be made.

Let me show you how to change our dinner menu by inserting 'VANILLA' as the first choice of ice creams.

```
EDIT C:  (I )nsert C:  (St )atement to follow A:  (3B)(<CR>);
 L:  (d)(<CR>);
 T:  (VANILLA)(<CR>);
EDIT C:  (Do )ne OK:  (<CR>);
```

I typed only 'I St 3B <CR>' to create the first line above. (I could, of course, have used the content address "ICE" instead of '3B' if I preferred.) You're already familiar with the C: and A: prompts which this line contains.

The L: prompt indicates that NLS needs to know whether an adjustment in level is needed. That is, whether the new statement is to be at the same level as the one before it, or whether one must go up or down to reach the level desired for the new statement. In this case I typed 'd <CR>' for down to indicate that the new statement is to be after statement 3B one level down. That is, it is to be numbered 3B1.

The T: prompt, as you know, means that NLS is expecting some text. In this case I typed 'VANILLA <CR>', the content of the statement to be inserted.

QUESTIONS?

**

When using the Insert command, you must remember to type a <CR> after you type the address, the level adjustment, and the text.

In all the tasks which follow, you may use either a statement number or a content address after the A: prompt. You will probably make fewer errors, however, if you stick to the statement numbers. (If you make an insertion you don't intend, you can use the Delete command to remove your error.)

Please practice the Insert command yourself by inserting APPLE as the first choice of pies.

TASK 22

```
EDIT C:  Insert C:  Statement to follow A:  3A;
 L:  d;
 T:  APPLE;
EDIT C:  Done OK: _;
```

Please print the file again to see whether this insertion has been made properly, and how the statement numbers have again been changed. Since only the desserts have been altered, please start printing at statement 3.

TASK 23

```
EDIT C:  Jump to A:  3;
EDIT C:  Print OK:/C: _;
   3  DESSERT
      3A  PIE
          3A1  APPLE
          3A2  BLUEBERRY
      3B  ICE CREAM
          3B1  VANILLA
          3B2  PEPPERMINT
          3B3  MAPLENUT
          3B4  CHOCOLATE
EDIT C:  Done OK: _;
```

Fine. As you can see, NLS has inserted a new statement 3A1 with the content "APPLE" and a new statement 3B1 with the content "VANILLA". It has renumbered (´demoted´) the statements which follow at the same level.

QUESTIONS?

**

Let´s see now how to add a list of beverages to the end of our dinner menu. Suppose we begin by inserting the heading BEVERAGE as a new statement right after DESSERT and at the same level with SOUP, ENTREE, and DESSERT.

What will its statement number be?

4

YOU ARE CORRECT.

You can cause NLS to assign statement number 4 to the new statement by inserting it after 3 and at the same level. You indicate ´same´ by typing a <CR> after the L: prompt to indicate that no level adjustment is needed. Make a new statement 4 now with the content BEVERAGE.

TASK 24

```
EDIT C:  Insert C:  Statement to follow A:  3;
 L: _;
 T:  BEVERAGE;
EDIT C:  Done OK: _;
```

The statement you've just inserted is the current statement, so you can see if you've inserted it correctly by simply typing '\' followed by a space. Do that now.

TASK 25

```
EDIT C:  \
 4  BEVERAGE
EDIT C:  Done OK:__;
```

Good. There are two other ways in which you could have specified this new statement with exactly the same result. You could have inserted a statement after 3B one level up, or after 3B4 two levels up. 'Up' is indicated by typing 'u <CR>' after the L: prompt. More than one up or down is specified by typing the desired number of u's or d's before typing a <CR>.

QUESTIONS?

**

Now let's insert the actual beverages themselves -- TEA and COFFEE. These are to be the second-level substatements of statement 4; they are to acquire the statement numbers 4A and 4B. See if you can insert these two statements yourself without further instruction. Print each statement directly after you have entered it to check what you have done.

Insert now the first beverage, TEA, and print the result.

TASK 26

```
EDIT C:  Insert C:  Statement to follow A:  4;
 L:  d;
 T:  TEA;
EDIT C:  \
 4A  TEA
EDIT C:  Done OK:__;
```

Good. Now insert the second beverage, COFFEE, and print it also.

TASK 27

```
EDIT C:  Insert C:  Statement to follow A:  4A;
 L: _;
 T:  COFFEE;
EDIT C:  \
 4B  COFFEE
EDIT C:  Done OK:__;
```

Fine. Note that the text we've been inserting has been so short that it fits easily on a single line. However, statements may occupy approximately 25 lines since they may contain up to 2000 characters. When typing in some long text, do not hit a <CR> when you come to the end of a line as that will terminate the Insert command. Just keep typing and NLS will automatically return the carriage to the beginning

of the next line for you.

Please print the entire file once more so you can see its completed structure.

TASK 28

```
EDIT C:  Jump to A:  0;
EDIT C:  Print OK:/C: _;
  <NLS-SCHOLAR>DINNER.LNLS;2  1-OCT-74  08:28  CLH  ;
   1  SOUP
      1A  VEGETABLE
      1B  CREAM OF MUSHROOM
   2  ENTREE
      2A  FRIED CHICKEN
      2B  PRIME RIBS
      2C  SALMON
          2C1  WITH CREAM SAUCE
   3  DESSERT
      3A  PIE
          3A1  APPLE
          3A2  BLUEBERRY
      3B  ICE CREAM
          3B1  VANILLA
          3B2  PEPPERMINT
          3B3  MAPLENUT
          3B4  CHOCOLATE
   4  BEVERAGE
      4A  TEA
      4B  COFFEE
EDIT C:  Done OK: _;
```

Please update this file to incorporate our changes into its main copy.

TASK 29

```
EDIT C:  Update C:  File OK:/C: _;
EDIT C:  Done OK: _;
```

QUESTIONS?

**

CREATING A FILE

Now that the dinner menu is completed, I'd like to show you a menu for breakfast. To see its content, load and print the file BREAKFAST.

TASK 30

```
EDIT C:  Load C:  File T:  BREAKFAST;
( NLS-SCHOLAR, BREAKFAST.LNLS;1, )
EDIT C:  Print OK:/C: _;
  <NLS-SCHOLAR>BREAKFAST.LNLS;1  1-OCT-74  03:50  CLH  ;
    1  JUICE
       1A  ORANGE
       1B  GRAPEFRUIT
```

```
    2  CEREAL
       2A  OATMEAL
           2A1  WITH RAISINS
       2B  CREAM OF WHEAT
       2C  CORN FLAKES
    3  EGGS
       3A  SCRAMBLED
       3B  FRIED
           3B1  SUNNY-SIDE-UP
           3B2  OVER-EASY
       3C  BOILED
    4  BEVERAGE
       4A  HOT CHOCOLATE
       4B  TEA
           4B1  WITH LEMON
           4B2  WITH SUGAR AND CREAM
       4C  COFFEE
  EDIT C:  Done OK:  ;
```

Please tear off the paper here, so you can refer to this file in what follows.

The DINNER file has been automatically stored away and BREAKFAST is now the current file.

Your next job is to create a new file named MYBREAKFAST, and to insert statements into it until it's the same as the BREAKFAST file which you just printed.

To create a new file, use the Create command. Specify that the name of the file is to be MYBREAKFAST. Please do that now.

TASK 31

```
EDIT C:  Create C:  File T:  MYBREAKFAST;
( NLS-SCHOLAR, MYBREAKFAST.LNLS;1, )
EDIT C:  Done OK:_;
```

Good. Now the file BREAKFAST has automatically been stored away and the file MYBREAKFAST has become the current file.

QUESTIONS?

**

INSERTING STATEMENTS IN A NEW FILE

Although this file is new, it is not truly empty. To see its content, use the Print command.

TASK 32

```
EDIT C:  Print OK:/C:_;
  <NLS-SCHOLAR>MYBREAKFAST.LNLS;1  1-OCT-74  08:30  CLH  ;
EDIT C:  Done OK:_;
```

You can see that NLS has already provided a statement zero for this file and filled it with identifying information. This is always done automatically whenever a new file is created.

The existence of statement zero is important since it provides an initial statement in the file after which an insertion can be made. Remember that when inserting a statement you must specify an already existing statement which your insertion is to follow. You must always start a new file by inserting statement 1 after statement zero, because statement zero is the only statement in the file.

Since statement zero is at level zero, the highest level of the file, what should you type after the L: prompt of an Insert command to insert statement 1 after statement zero?

d

VERY GOOD.

Start now to copy the BREAKFAST file by inserting statement 1 with the content "JUICE" in the MYBREAKFAST file.

TASK 33

```
EDIT C:  Insert C:  Statement to follow A:  0;
 L:  d;
 T:  JUICE;
EDIT C:  Done OK: _;
```

Good. Your file now contains statement 0 and statement 1, which has the content "JUICE". Proceed to insert the next first-level heading, "CEREAL", as statement 2. Print the statement as soon as you've inserted it by using the '\' command.

TASK 34

```
EDIT C:  Insert C:  Statement to follow A:  1;
 L: _;
 T:  CEREAL;
EDIT C:  \
 2  CEREAL
EDIT C:  Done OK: _;
```

Fine. Now insert the remaining headings, "EGGS" and "BEVERAGE", as statements 3 and 4. Print each one as soon as you've inserted it.

TASK 35

```
EDIT C:  Insert C:  Statement to follow A:  2;
 L: _;
 T:  EGGS;
EDIT C:  \
 3  EGGS
EDIT C:  Insert C:  Statement to follow A:  3;
 L: _;
 T:  BEVERAGE;
```

```
EDIT C:  \
 4  BEVERAGE
EDIT C:  Done OK: _;
```

QUESTIONS?

**

REPEAT MODE

OK. Now let's go back and fill in the lower level statements of the file. This task presents no new problems and constitutes a review of what you've learned earlier.

But again, I'd like to confess that you did more typing than was necessary. If you wish to use the same command repeatedly, you can terminate the command by typing <CTRL-B> instead of <CR>. This puts you into what is called 'repeat mode'. Let me show you how that works with the Insert command by inserting the two juices, "ORANGE" and "GRAPEFRUIT", as statements 1A and 1B.

```
EDIT C:  (I )nsert C:  (St )atement to follow A:  (1)(<CR>);
 L:  (d)(<CR>);
 T:  (ORANGE)(<CTRL-B>)^B A:  (1A)(<CR>);
 L:  (<CR>);
 T:  (GRAPEFRUIT)(<CR>);
EDIT C:  (Do )ne OK:  (<CR>);
```

At the end of the first text, "ORANGE", I typed <CTRL-B> instead of <CR>. (Note that this echoes as ^B.) This indicated that I wanted to repeat the Insert command. So NLS proceeded as if I had typed 'I St ' and gave me an A: prompt asking for the address. I then proceeded as usual, typing a <CR> at the end of the second text, "GRAPEFRUIT".

QUESTIONS?

**

Try using repeat mode to insert the remaining statements of branch 2. Statement 2, with content "CEREAL", already exists in the MYBREAKFAST file. Proceed to insert after it all the statements under it, reading them from the BREAKFAST file. Remember to type <CTRL-B> rather than <CR> to terminate every text but the last. When you've entered the end statement of the branch, statement 2C, type <CR> to return to the EDIT herald. Please begin now.

TASK 36

```
EDIT C:  Insert C:  Statement to follow A:  2;
 L:  d;
 T:  OATMEAL^B A:  2A;
 L:  d;
 T:  WITH RAISINS^B A:  2A1;
 L:  u;
 T:  CREAM OF WHEAT^B A:  2B;
 L:  _;
 T:  CORN FLAKES;
EDIT C:  Done OK: _;
```

Great. Please print your file starting with statement 2 to see what it looks like.

TASK 37

```
EDIT C:  Jump to A:  2;
EDIT C:  Print OK:/C: _ ;
    2  CEREAL
       2A  OATMEAL
           2A1  WITH RAISINS
       2B  CREAM OF WHEAT
       2C  CORN FLAKES
    3  EGGS
    4  BEVERAGE
EDIT C:  Done OK: _;
```

QUESTIONS?

**

CURRENT ADDRESS

Let's procced with the MYBREAKFAST file by inserting the remaining statements of branch 3. This task is virtually the same as that of inserting the statements of branch 2, but let me show you one more way of making it even shorter.

Instead of typing a statement number or a content address every time an A: prompt appears, you can often type just a <CR>. This indicates that the address you want is that of the current statement. Since one often inserts statements in sequential order, the address you want to follow is usually the current address. Let me show you how this works.

```
EDIT C:  (I )nsert C:  (St )atement to follow A:  (3)(<CR>);
 L:  (d)(<CR>);
 T:  (SCRAMBLED)(<CTRL-B>)^B A:  (<CR>); L:  (<CR>);
 T:  (FRIED)(<CTRL-B>)^B A:  (<CR>); L:  (d)(<CR>);
 T:  (SUNNY-SIDE-UP)(<CTRL-B>)^B A:  (<CR>); L:  (<CR>);
 T:  (OVER-EASY)(<CTRL-B>)^B A:  (<CR>); L:  (u)(<CR>);
 T:  (BOILED)(<CR>);
EDIT C:  (Do )ne OK:  (<CR>);
```

After every A: prompt except the first I simply typed <CR> to indicate that each new insertion is to follow the one before. When the address is specified in this manner, NLS puts the L: prompt on the same line, changing the format so that the T: prompt is always at the left.

Print the file starting at statement 3 to see that these insertions have been made properly.

TASK 38

```
EDIT C:  Jump to A:  3;
EDIT C:  Print OK:/C: _ ;
    3  EGGS
       3A  SCRAMBLED
       3B  FRIED
           3B1  SUNNY-SIDE-UP
           3B2  OVER-EASY
       3C  BOILED
```

```
    4  BEVERAGE
EDIT C:  Done OK: _;
```

QUESTIONS?

**

Try using the current address in conjunction with repeat mode to finish the MYBREAKFAST file by inserting the remaining statements of branch 4.

TASK 39

```
EDIT C:  Insert C:  Statement to follow A:  4;
 L:  d;
 T:  HOT CHOCOLATE^B A: _; L: _;
 T:  TEA^B A: _; L:  d;
 T:  WITH LEMON^B A: _; L: _;
 T:  WITH SUGAR AND CREAM^B A: _; L:  u;
 T:  COFFEE;
EDIT C:  Done OK: _;
```

Please update and then print the completed MYBREAKFAST file, starting with statement 0, so you can compare it with the BREAKFAST file.

TASK 40

```
EDIT C:  Update C:  File OK:/C: _;
EDIT C:  Jump to A:  0;
EDIT C:  Print OK:/C: _;
  <NLS-SCHOLAR>MYBREAKFAST.LNLS;2  1-OCT-74  08:55  CLH  ;
    1  JUICE
       1A  ORANGE
       1B  GRAPEFRUIT
    2  CEREAL
       2A  OATMEAL
           2A1  WITH RAISINS
       2B  CREAM OF WHEAT
       2C  CORN FLAKES
    3  EGGS
       3A  SCRAMBLED
       3B  FRIED
           3B1  SUNNY-SIDE-UP
           3B2  OVER-EASY
       3C  BOILED
    4  BEVERAGE
       4A  HOT CHOCOLATE
       4B  TEA
           4B1  WITH LEMON
           4B2  WITH SUGAR AND CREAM
       4C  COFFEE
EDIT C:  Done OK: _;
```

You can see that the files are identical except for the information in statement 0.

QUESTIONS?

**

SUBSTITUTE WORD IN STATEMENT

Before we end this lesson, I'd like to show you how to change the content of statements which have already been inserted in your file.

You've learned how to correct errors by deleting an entire statement and inserting a new one in its place. The Substitute command may be used to change just a word or even a single character at a time. For example, here's how you can change a word in statement 2A1 so that it reads "WITH DATES" instead of "WITH RAISINS".

```
EDIT C: (Su )bstitute C: (W )ord in  C: (S )tatement at  A: (2A1)(<CR>);
 <New WORD> T: (DATES)(<CR>);
 <Old WORD> T: (RAISINS)(<CR>);
Finished? Y/N:  OK: (<CR>);
Substitutions made: 1
EDIT C: (Do )ne OK: (<CR>);
```

You can see that I've specified that I want to substitute one word in statement 2A1 for another; the new word "DATES" is to be substituted for the old word "RAISINS". (There is no restriction that the items substituted for one another be of the same length.)

To do this I typed only 'Su W S 2A1 <CR>' to produce the first line; I typed 'DATES <CR>' and 'RAISINS <CR>' to produce the second and third. The remaining characters are all printed by NLS.

NLS asks Finished? since more than one substitution can be made at a time. If I had typed 'N' for no instead of 'Y' for yes, it would have prompted for another <New WORD> and <Old WORD> pair.

After I typed 'Y' -- which you won't see because it isn't echoed -- NLS asks for further confirmation with OK: to which I responded with a <CR>.

When the command has been executed, NLS tells how many substitutions have been made. If there had been two instances of RAISINS in statement 2A1, both would have been changed to read DATES.

Statement 2A1 is now the current statement. Please print it.

TASK 41

```
EDIT C: /
    ==>WITH DAT
EDIT C: ↑
 2A OATMEAL
EDIT C: %

 2A1 WITH DATES
EDIT C: Done OK: ;
```

You printed too many statements.
This is what I wanted you to print:
 2A1 WITH DATES

        Please try to print what
wanted. The file is as you left it.

continuation of TASK 41

EDIT C: *What command prints the current statement?*

 THE USER PRINTS THE CURRENT STATEMENT USING THE
\ COMMAND.

continuation of TASK 41

EDIT C: \
 2A1 WITH DATES
EDIT C: Done OK: ;

QUESTIONS?

**

Try the Substitute command yourself by substituting the word "OAT" for
"CORN" in statement 2C, "CORN FLAKES".

TASK 42

EDIT C: Substitute C: Word in C: Statement at A: 2C;
 <New WORD> T: CORN;
 <Old WORD> T: OAT;
Finished? Y/N: OK: ;
Substitutions made: 0
EDIT C: Done OK: ;

I wanted you to change
this part of your file:
    2B CREAM OF WHEAT
    2C CORN FLAKES
 3 . . .

into this:
    2B CREAM OF WHEAT
    2C OAT FLAKES
 3 . . .

But you did not make any changes to your file.

        Please try to finish this task.
You may continue where you left off.

continuation of TASK 42

```
EDIT C: Substitute C: Word in  C: Statement at  A: 2C;
 <New WORD> T: OAT;
 <Old WORD> T: CORN ;
Finished? Y/N: _ OK: _;
Substitutions made: 1
EDIT C: Done OK: _;
```

You should have altered the file,
so that this part of it

```
    2B CREAM OF WHEAT
    2C CORN FLAKES
 3 . . .
```

would look like this:

```
    2B CREAM OF WHEAT
    2C OAT FLAKES
 3 . . .
```

But you altered this part of it

```
    2B CREAM OF WHEAT
    2C CORN FLAKES
 3 . . .
```

to look like this:

```
    2B CREAM OF WHEAT
    2C OATFLAKES
 3 . . .
```

I have restored the file as it was
at the beginning of this task. Please start over.

TASK 42

```
EDIT C: Substitute C: Word in  C: Statement at  A: 2C;
 <New WORD> T: OAT;
 <Old WORD> T: CORN;
Finished? Y/N: _ OK: _;
Substitutions made: 1
EDIT C: Done OK: _;
```

Now print your result.

TASK 43

```
EDIT C: \
 2C OAT FLAKES
EDIT C: Done OK: _;
```

Good. The noun 'WORD' refers to the fact that whole words are involved, not just parts of words; but it does not imply that a single word only may be provided as either the <New WORD> or the <Old WORD>.

Use the Substitute command to change statement 2A1 again, this time to read 'WITH BROWN SUGAR AND CREAM' instead of 'WITH DATES' and print your result.

TASK 44

```
EDIT C: Substitute C: Word in  C: Statement at  A: 2A1;
 <New WORD> T: WITH BROWN SUGAR AND CREAM;
 <Old WORD> T: DATES;
Finished? Y/N:__OK:_;
Substitutions made: 1
EDIT C: Done OK:_;
```

What I wanted you to do
should have transformed this:

```
    2A OATMEAL
       2A1 WITH DATES
    2B . . .
```

into this:

```
    2A OATMEAL
       2A1 WITH BROWN SUGAR AND CREAM
    2B . . .
```

But what you did transformed this:

```
    2A OATMEAL
       2A1 WITH DATES
    2B . . .
```

into this:

```
    2A OATMEAL
       2A1 WITH WITH BROWN SUGAR AND CREAM
    2B . . .
```

Please try again. I have
restored the file to its original state.

TASK 44

EDIT C: *How do I do this confusing task?*

THE USER TYPES 'SUBSTITUTE ', FOLLOWED BY 'WORD ', 'STATEMENT ', '2A1', <CR>, 'BROWN SUGAR AND CREAM', <CR>, 'DATES', <CR>, 'Y', <CR>, '\', A <SPACE>, 'DONE ', AND <CR>.

continuation of TASK 44

EDIT C: *Please do it for me*

```
EDIT C: (Substitute ) C: (Word ) in  C: (Statement ) at  A: (2A1
)(<CR>);
 <New WORD> T: (BROWN SUGAR AND CREAM)(<CR>);
 <Old WORD> T: (DATES)(<CR>);
Finished? Y/N:  OK: (<CR>);
Substitutions made: 1
EDIT C: (\ )
 2A1 WITH BROWN SUGAR AND CREAM
EDIT C: (Done ) OK: (<CR>);
```

QUESTIONS?

**

SUBSTITUTE CHARACTER IN STATEMENT

Another noun which may follow the verb 'Substitute' is 'Character'. Thus one may make substitutions in terms of a single character or a string of characters as well as in terms of words. Watch what happens when I substitute the characters "EA" for "U" in statement 2B, CREAM OF WHEAT.

```
EDIT C: (Su )bstitute C: (C )haracter in  C: (S )tatement at  A: (2B
)(<CR>);
 <New CHARACTER> T: (U)(<CR>);
 <Old CHARACTER> T: (EA)(<CR>);
Finished? Y/N:  OK: (<CR>);
Substitutions made: 2
EDIT C: (\ )
 2B CRUM OF WHUT
EDIT C: (Do )ne OK: (<CR>);
```

QUESTIONS?

**

SUBSTITUTE CHARACTER IN BRANCH

Rather than substituting a word or character in just a single statement you may make the substitution for all statements of a branch. Try that now by substituting asterisks (*) for all the hyphens (-) in branch 3B.

TASK 45

```
EDIT C: Substitute C: Character in  C: Branch at  A: 3B;
 <New CHARACTER> T: *;
 <Old CHARACTER> T: -;
Finished? Y/N: _OK:_;
Substitutions made: 3
EDIT C: Done OK:_;
```

You should have altered the file,
so that this part of it

```
     3B FRIED
        3B1 SUNNY-SIDE-UP
        3B2 OVER-EASY
     3C . . .
```

would look like this:

```
     3B FRIED
        3B1 SUNNY*SIDE*UP
        3B2 OVER*EASY
     3C . . .
```

But you altered this part of it

```
     3B FRIED
        3B1 SUNNY-SIDE-UP
        3B2 OVER-EASY
     3C . . .
```

to look like this:

```
     3B FRIED
        3B1 SUNNY:SIDE:UP
        3B2 OVER:EASY
     3C . . .
```

I have restored the file as it was at the beginning of this task. Please start over.

TASK 45

EDIT C: *Do this task,please*

```
EDIT C: (Substitute ) C: (Character ) in  C: (Branch ) at  A: (3B
)(<CR>);
 <New CHARACTER> T: (*)(<CR>);
 <Old CHARACTER> T: (-)(<CR>);
Finished? Y/N:  OK: (<CR>);
Substitutions made: 3
EDIT C: (Done ) OK: (<CR>);
```

The message 'Substitute in Progress' is printed by NLS to indicate that it is busy working. There is no practical limit to the number of substitutions which may occur as the result of a single Substitute command. Using branch 0 as the address results in the substitution(s) being made over the entire file.

Please print the file starting with statement 1 so you can see the effect of the changes we have made.

TASK 46

EDIT C: *Tell me how to print the file*

THE USER PRINTS AN NLS FILE FROM THE CURRENT STATEMENT

TO THE END OF AN NLS FILE USING THE PRINT COMMAND,
 PROCEDURE: HE TYPES 'PRINT ' FOLLOWED BY <CR>.

continuation of TASK 46

EDIT C: *what does current statement mean?*

 THE CURRENT STATEMENT IS THE STATEMENT AT
 WHICH THE CM IS CURRENTLY POINTING.

continuation of TASK 46

EDIT C: *Where is the control marker now?*

THE CURRENT POSITION OF THE CM IS:
 = 3B + 1

continuation of TASK 46

EDIT C: *How do I move the CM?*

 THE USER MOVES THE CM USING A CM MOVING COMMAND.

continuation of TASK 46

EDIT C: *What are the CM moving commands?*

 EXAMPLES OF A CM MOVING COMMAND ARE THE JUMP COMMAND,
THE _ COMMAND, AND THE > COMMAND.

continuation of TASK 46

EDIT C: *What does the jump command do?*

 THE JUMP COMMAND MOVES THE CM.

continuation of TASK 46

EDIT C: Jump to A: 1;
EDIT C: Print OK:/C: _;
    1 JUICE
       1A ORANGE

```
      1B GRAPEFRUIT
   2 CEREAL
      2A OATMEAL
         2A1 WITH BROWN SUGAR AND CREAM
      2B CRUM OF WHUT
      2C OAT FLAKES
   3 EGGS
      3A SCRAMBLED
      3B FRIED
         3B1 SUNNY*SIDE*UP
         3B2 OVER*EASY
      3C BOILED
   4 BEVERAGE
      4A HOT CHOCOLATE
      4B TEA
         4B1 WITH LEMON
         4B2 WITH SUGAR AND CREAM
      4C COFFEE
EDIT C: Done OK: ;
```

QUESTIONS?

How do I print just branch 3?

THE USER PRINTS BRANCH 3 USING THE PRINT COMMAND,
PROCEDURE: HE TYPES 'PRINT ', FOLLOWED BY
'BRANCH ', '3', <CR>, VIEWSPECS, AND <CR>.

**

THE QUESTION MARK FACILITY

You can now practice further with the Substitute command, trying different combinations of command terms and making more than one substitution at a time by typing N instead of Y after FINISHED?.

You may also wish to practice further using content addressing, and to insert, delete, and create new files.

As you do this, you may benefit from using the question mark facility. After any C: prompt you may type '?' and NLS will provide you with a list of all command terms which are valid at that point. You may then type one of them and proceed.

If you type '?' after an EDIT C: you will be presented with a complete list of NLS verbs; this is probably more information than you desire.

We will use the BREAKFAST file in Lesson 3, so that should not be modified. However you may change the MYBREAKFAST file, which is the current file, in any way that you wish. Please play with the file, trying out the various commands you have learned. Type 'Done <CR>' as usual when you're all finished.

TASK 47

EDIT C: Done OK:_;

QUESTIONS?

NLS

...USING A DISPOSABLE COPY OF YOUR FILE

```
EDIT C: Print OK:/C: _C:_
CURRENT ALTERNATIVES ARE:
Branch              Statement           Plex                Group
----_##
EDIT C: S
CURRENT ALTERNATIVES ARE:
Substitute          Set                 Show
----_##
EDIT C: D
CURRENT ALTERNATIVES ARE:
Delete              Done
----_##
EDIT C: Substitute C: _
CURRENT ALTERNATIVES ARE:
Character           Word                Visible             Invisible
Text
----_##
EDIT C: Done OK:_;
```

...DISPOSING OF THIS COPY

**

REVIEW OF LESSON TWO

Lesson Two has introduced the following commands:

```
Insert command
Create command
Substitute command for words in statements
                   for characters in statements
                   for characters in branches
```

QUESTIONS?

**

It has also introduced the following concepts:

```
echoing
field
level
level zero
```

```
L:  prompt
level adjustment
   up (u)
   down (d)
   same (<CR>)

repeat mode
<CTRL-B>
question mark facility

QUESTIONS?
```

What is a field?

A FIELD IS A SEQUENCE OF CONTIGUOUS LETTERS OR CONTIGUOUS DIGITS WITHIN A STATEMENT NUMBER.

What statements are at level 2?

THE STATEMENTS AT LEVEL 2 ARE:
1A 1B 2A 2B 2C 3A 3B 3C 4A 4B 4C

What is the level of statement 14AC3?

THE LEVEL OF STATEMENT 14AC3 IS:
3

What is the level of statement zero?

THE LEVEL OF STATEMENT 0 IS:
0

Define repeat mode

REPEAT MODE IS A METHOD OF REPEATING THE INSERT COMMAND WITHOUT HAVING TO DO EXTRA TYPING.

How do I use repeat mode?

THE USER REPEATS THE INSERT COMMAND USING
REPEAT MODE,
PROCEDURE: HE TYPES <CTRL-B> AFTER THE INSERT COMMAND.

**

Lesson Three will introduce selective printing, viewspecs, groups, plexes, and commands which move and copy structures. Au revoir.

LESSON THREE

REVIEW OF PRINTING COMMANDS

Hello. Welcome back.

I'd like to start this lesson by reviewing what you know about printing.

In Lesson One you learned how to move the CM with the Jump command, and to print the current statement by typing a one-character command. What character prints the current statement?

**

Right. What character prints the back statement?

↑

Good. What character prints the next statement?

%

That's right. You also learned how to print the entire content of a file by moving the CM to statement zero and typing simply 'Print <CR>'. If you wanted to print only part of a file you moved the CM to the place at which you wanted to begin; then 'Print <CR>' caused the file to be printed from there to the end.

QUESTIONS?

**

MODIFYING THE PRINT COMMAND

Now I'd like to show you how to use the Print command to produce more selective printing.

Instead of typing a <CR> directly after 'Print', you may type a noun specifying the kind of structure unit you want printed. Thus you may print a single statement by typing 'Print Statement' or an entire branch by typing 'Print Branch'. Let me show you how this works by printing branch 3 of the BREAKFAST file.

Please load the BREAKFAST file for me so I can begin.

TASK 48

```
EDIT C:  Load C:  File T:  BREAKFAST;
( NLS-SCHOLAR, BREAKFAST.LNLS;2, )
EDIT C:  Done OK: _ ;
```

Thank you. Now I'll print branch 3.

```
EDIT C:  (Pri )nt OK:/C:  C:  (B )ranch at A:  (3)(<CR>);
 V:  (<CR>);
 3  EGGS
    3A  SCRAMBLED
    3B  FRIED
```

```
        3B1  SUNNY-SIDE-UP
        3B2  OVER-EASY
    3C  BOILED
EDIT C:  (Do )ne OK:  (<CR>);
```

NLS prompts with OK:/C: to indicate that you may type either a <CR> or a command term. When I typed 'B' for branch, it echoed the C: prompt again to indicate that I had chosen a command term. It then prompts with A: for address because it needs to know which branch is to be printed.

The V: prompt stands for 'viewspecs' which I'll describe presently. I typed a <CR> to indicate that no change in viewspecs was needed.

Use the Print command yourself now to print branch 3B, a sub-branch of branch 3 consisting of all statements whose numbers begin with 3B. For the present, just type a <CR> whenever the V: prompt appears.

TASK 49

```
EDIT C:  Print OK:/C:  C:  Branch at A:  3B;
 V: _;
 3B  FRIED
     3B1  SUNNY-SIDE-UP
     3B2  OVER-EASY
EDIT C:  Done OK: _;
```

Good. Similarly you can print just a single statement at a time by using 'Statement' as the noun following 'Print'. Try that now by printing statement 3B1.

TASK 50

```
EDIT C:  Print OK:/C:  C:  Statement at A:  3B1;
 V: _;
 3B1  SUNNY-SIDE-UP
EDIT C:  Done OK: _;
```

Note that this gives the same effect as 'Jump 3B1' followed by the '\' command.

QUESTIONS?

**

VIEWSPECS

The V: prompt appears whenever you modify the Print command. It asks whether you wish to change the 'viewspecs'. Viewspecs control the way in which you view a file. For example, they control whether the statement numbers are printed or not, whether indenting is done or not, whether only first level statements of the file are shown or all statements, whether blank lines are printed between statements or not, etc.

NLS provides a standard set of 'default' viewspecs which are suitable for most purposes. It also allows the user to specify the viewspecs which he would like to have as the standard ones and to change these

standard viewspecs during the course of his work. For this primer I've chosen to set the standard viewspecs so that statement numbers are printed to the left of each statement, 3 spaces are indented for each level, and all lines and all levels of the file appear, and no blank lines are printed between statements.

When you type just a <CR> after a V: prompt, it means that you are satisfied with the standard viewspecs. If you want to change them for this command only, you type a sequence of one or more 'viewspecs' after the V: prompt. Each viewspec controls a different aspect of the appearance of the file. Note that the file itself is not modified, just your view of it.

The viewspecs are arranged in alphabetically contiguous pairs. The first of the pair specifies that a particular viewspec feature is to be 'on', while the second specifies 'off'. For example, viewspec A turns indenting on, while viewspec B turns indenting off.

To print branch 3 with statement numbers off (viewspec n) and indenting off (viewspec B), you could do the following:

```
EDIT C:  (Pri )nt OK:/C:  C:  (B )ranch at A:  (3)(<CR>);
 V:  (Bn)(<CR>);
  EGGS
  SCRAMBLED
  FRIED
  SUNNY-SIDE-UP
  OVER-EASY
  BOILED
EDIT C:  (Do )ne OK:  (<CR>);
```

You can see that the appearance of the file is changed considerably even though the content has remained the same.

Viewspec B and viewspec n remain in effect for this command only. They are automatically changed back to the standard viewspecs -- viewspec A, indenting on, and viewspec m, statement numbers on --- as soon as the printing is completed. (Note that m and n are lower case characters -- you will not be aware of this if you are working with an upper case only terminal.)

You can see that the standard viewspecs have not been changed if you print branch 3 again, typing <CR> after the V: prompt to indicate the standard viewspecs. Please do that now.

TASK 51

```
EDIT C:  Print OK:/C:  C:  Branch at A:  3;
 V: _;
 3  EGGS
    3A  SCRAMBLED
    3B  FRIED
        3B1  SUNNY-SIDE-UP
        3B2  OVER-EASY
    3C  BOILED
EDIT C:  Done OK: _;
```

QUESTIONS?

**

VIEWING SELECTED LEVELS

A more powerful use of viewspecs involves viewing certain levels of the file selectively. For example, to gain a general idea of a file's content, you could view only the first-level statements. This is done with viewspec x, which causes only the first line of each first-level statement to appear. (Here all our statements are so short that they occupy only one line anyway. But remember that statements may contain up to 2000 characters.) The standard viewspec is its companion, viewspec w, show all lines and all levels.

Print branch 0 now -- the entire file -- with viewspec x in effect. (Note that this is a lower case 'x'. If you are using an upper case only terminal, you must use the shift character '!' before the 'x' to indicate lower case -- that is, you must type '!x', but only a capital X will be echoed.)

TASK 52

```
EDIT C:  Print OK:/C:  C:  Branch at A:  0;
 V:  x;
  <NLS-SCHOLAR>BREAKFAST.LNLS;1  01-OCT-74  08:20  CLH  ;
   1  JUICE
   2  CEREAL
   3  EGGS
   4  BEVERAGE
EDIT C:  Done OK: _;
```

Viewspec b, show one level more, may be used to increase the number of levels which are visible. Each instance of a 'b' causes one more level to be added. Print the entire file again, showing levels 1 and 2 by using viewspec b in conjunction with viewspec x. Note that you will have to type the 'x' before the 'b' since the levels must first be set to one and then increased. This is one of the few cases in which the order in which the viewspecs are specified is of importance. (Note also that this is a lower case 'b' so remember to use the shift character '!' before each 'b' if you are working with an upper case only terminal.)

TASK 53

```
EDIT C:  Print OK:/C:  C:  Branch at A:  0;
 V:  xb;
  <NLS-SCHOLAR>BREAKFAST.LNLS;1  01-OCT-74  08:20  CLH  ;
   1  JUICE
      1A  ORANGE
      1B  GRAPEFRUIT
   2  CEREAL
      2A  OATMEAL
      2B  CREAM OF WHEAT
      2C  CORN FLAKES
   3  EGGS
      3A  SCRAMBLED
      3B  FRIED
      3C  BOILED
   4  BEVERAGE
```

```
      4A  HOT CHOCOLATE
      4B  TEA
      4C  COFFEE
EDIT C:  Done OK: _;
```

Very good. If you had typed 'xbb' then three levels of the file would have been shown.

QUESTIONS?

**

THE SET VIEWSPECS COMMAND

To change the standard viewspecs so they will remain in effect for more than one command, you use the Set Viewspecs command. The viewspecs which you supply following the V: prompt of this command will remain in effect until you change them with another Set Viewspecs command. I'll change the standard viewspecs now to show one level only, turn statement numbers off, and turn blank lines on.

```
EDIT C: (Se )t C: (V )iewspecs  V: (xny)(<CR>);
EDIT C: (Do )ne OK: (<CR>);
```

Viewspec y turns blank lines on, while its companion viewspec z turns blank lines off. These are both lower case characters.

Now to show the effect of my changing the standard viewspecs, print the whole BREAKFAST file again, using <CR> after the V: prompt to indicate that you want the standard viewspecs.

TASK 54

```
EDIT C: Print OK:/C:  C: Branch at A: 0;
 V: _;
  <NLS-SCHOLAR>BREAKFAST.LNLS;2,  21-MAY-74 21:37 AGB2 ;

     JUICE

     CEREAL

     EGGS

     BEVERAGE

EDIT C: Done OK: _;
```

Good. Now print branch 4 without changing any viewspecs to see that these viewspecs persist through more than one command.

TASK 55

```
EDIT C: Print OK:/C:  C: Branch at A: 4;
 V: m;
 4 BEVERAGE
EDIT C: Done OK: _;
```

With regard to the V: viewspecs,
I did not want you to change the V: viewspecs;
but you set the V: viewspecs to m.

Please try again. I have
restored the file to its original state.

TASK 55

EDIT C: *What are the current viewspecs?*

THE CURRENT VIEWSPECS ARE:
levels: 1 , lines: 1 , hjnyACFHJLP

continuation of TASK 55

EDIT C: *What does viewspec m do?*

VIEWSPEC m TURNS STATEMENT NUMBERS ON.

continuation of TASK 55

EDIT C: Print OK:/C: C: Branch at A: 4;
V: _;
BEVERAGE
EDIT C: Done OK: _;

Now please change the standard viewspecs back the way they were:
-- show all levels
-- turn statement numbers on
-- turn blank lines off
so that we can proceed.

TASK 56

EDIT C: *How do I turn blank lines off?*

VIEWSPEC z TURNS BLANK LINES OFF.

continuation of TASK 56

EDIT C: Set C: Viewspecs V: z;
EDIT C: Done OK: _;

With regard to the standard viewspecs,
        I wanted you to turn the blank lines off,
        to turn the statement numbers on,
        to show all levels,
        and to show all lines;
but you only turned the blank lines off.
That is not sufficient.

        Please try to repair your
mistake. The file is as you left it.

continuation of TASK 56

EDIT C: *What viewspec turns statement numbers on?*

 VIEWSPEC m TURNS STATEMENT NUMBERS ON.

continuation of TASK 56

EDIT C: Set C: Viewspecs V: mcr;
EDIT C: Done OK:_;

With regard to the standard viewspecs,
        I wanted you to turn the blank lines off,
        to turn the statement numbers on,
        to show all levels,
        and to show all lines;
but instead you turned the blank lines off,
        turned the statement numbers on,
        showed all levels,
        and showed 2 lines.

        Please try to finish this task.
You may continue where you left off.

continuation of TASK 56

EDIT C: *What viewspec shows all lines?*

 THE USER SHOWS ALL LINES USING VIEWSPEC w.

continuation of TASK 56

EDIT C: Set C: Viewspecs V: w;
EDIT C: Done OK:_;

Though I was expecting something slightly different, your answer is also correct.

QUESTIONS?

**

THE SHOW VIEWSPECS COMMAND

A common error in using NLS occurs when the student thinks some of his statements are missing. He supposes they have not been inserted properly or have somehow been deleted. Actually they may be temporarily invisible because of the viewspec setting. To determine the current setting of the viewspecs, use the Show Viewspecs command. Please try that now to see what is printed.

TASK 57

EDIT C: Show C: Viewspecs
levels: ALL, lines: ALL, hjmzACFHJLP
EDIT C: Done OK:_ ;

The 'levels: ALL, lines: ALL' means that viewspec w is in effect. (If viewspec x had been in effect, the message 'levels: 1, LINES: 1' would have appeared.) The remaining viewspecs are simply listed in alphabetical order with the lower case ones preceding the upper case ones.

Each viewspec that is printed has a companion viewspec which is not printed because it is not in effect at this time. In addition, some viewspec pairs are not printed at all. For example, neither viewspec b, show one level more, nor its companion viewspec a, show one level less, appears here because information about their setting is represented in the value of LEVELS.

As you can see, there are a great many viewspecs. If you are interested in what they control you may look them all up on the NLS Cue Card. However, the ones that have been introduced here are likely to be sufficient for most purposes.

QUESTIONS?

**

Now that the viewspecs are set to the original standard viewspecs, please print the entire BREAKFAST file again so we can refer to it in what follows.

TASK 58

EDIT C: Print OK:/C: C: Branch at A: 0;
V:_ ;
<NLS-SCHOLAR>BREAKFAST.LNLS;1 01-OCT-74 08:20 CLH ;
1 JUICE
1A ORANGE

```
      1B  GRAPEFRUIT
   2  CEREAL
      2A  OATMEAL
          2A1  WITH RAISINS
      2B  CREAM OF WHEAT
      2C  CORN FLAKES
   3  EGGS
      3A  SCRAMBLED
      3B  FRIED
          3B1  SUNNY-SIDE-UP
          3B2  OVER-EASY
      3C  BOILED
   4  BEVERAGE
      4A  HOT CHOCOLATE
      4B  TEA
          4B1  WITH LEMON
          4B2  WITH SUGAR AND CREAM
      4C  COFFEE
EDIT C:  Done OK: _;
```

Please tear off the paper here for future reference.

THE MOVE COMMAND

You've seen at the end of Lesson Two how to edit a file by using the Substitute command to change the content of an individual statement or of all statements in a branch. Now I'd like to show you how to change the structure of your file by changing the position of the statements rather than their content.

The Move command causes a structure unit (statement, branch, plex, or group) to be moved from its present address to one which you specify. For example, if you wanted to make "BOILED" eggs the first kind listed under "EGGS" in your menu, that could be done as follows:

```
EDIT C:  (Mo )ve C:  (S )tatement from A:  (3C)(<CR>);
 to follow A:  (3)(<CR>);
 L:  (d)(<CR>);
EDIT C:  (Do )ne OK:  (<CR>);
```

Note that the method of specifying the new address is the same as that used by the Insert command; you must give the address to be followed and specify whether a change in level is needed.

Please print branch 3 now with the standard viewspecs so you can see how it has been altered.

TASK 59

```
EDIT C:  Print OK:/C:  C:  Branch at A:  3;
 V: _;
 3  EGGS
    3A  BOILED
    3B  SCRAMBLED
    3C  FRIED
        3C1  SUNNY-SIDE-UP
        3C2  EASY-OVER
EDIT C:  Done OK: _;
```

When a structure unit is moved it no longer appears at its old address, only at the new one.

QUESTIONS?

**

Suppose that you wanted to make "FRIED" the first kind of "EGGS" instead of the last. Try that now and see what happens.

TASK 60

```
EDIT C:  Move C:  Statement from A:  3C;
 to follow A:  3;
 L:  d;
ILLEGAL MOVE
EDIT C:  Done OK: _;
```

The message 'ILLEGAL MOVE' was printed because it is not possible to move a statement having any substatements. (This is similar to the situaton with the Delete command which you encountered in Lesson One. This message will also appear if you attempt to move statement zero.) In any event, the entire branch describing fried eggs should be moved, not just its first statement. Please do that now.

TASK 61

```
EDIT C:  Move C:  Branch from A:  3C;
 to follow A:  3;
 L:  d;
EDIT C:  Done OK: _;
```

Good. Now please print branch 3 once more to see its new appearance.

TASK 62

```
EDIT C:  Print OK:/C:  C:  Branch at A:  3;
 V: _;
 3  EGGS
    3A  FRIED
        3A1  SUNNY-SIDE-UP
        3A2  EASY-OVER
    3B  BOILED
    3C  SCRAMBLED
EDIT C:  Done OK: _;
```

QUESTIONS?

**

THE COPY COMMAND

The Copy command has the same effect as the Move command except that the structure unit which is copied appears at both the old address and the new.

Try the Copy command by copying the beverages to that they will appear both as the first branch of the file and the last.

TASK 63

```
EDIT C:  Copy C:  Branch from A:  4;
 to follow A:  0;
 L:  d;
EDIT C:  Done OK: _;
```

To see what has occurred, print the file showing one level only.

TASK 64

```
EDIT C:  Print OK:/C:  C:  Branch at A:  0;
 V:  x;
  <NLS-SCHOLAR>BREAKFAST.LNLS;1  01-OCT-74  08:20  CLH  ;
   1  BEVERAGE
   2  JUICE
   3  CEREAL
   4  EGGS
   5  BEVERAGE
EDIT C:  Done OK: _;
```

Now please delete that first branch since we don't really want the branch "BEVERAGE" twice on the menu.

TASK 65

```
EDIT C:  Delete C:  Branch at A:  1;
 OK: _;
EDIT C:  Done OK: _;
```

QUESTIONS?

**

THE TRANSPOSE COMMAND

Two structure units can be interchanged by using the Transpose command and specifying the two addresses involved. To see how this works, transpose the two statements "ORANGE" and "GRAPEFRUIT".

TASK 66

```
EDIT C:  Transpose C:  Statement at A:  1A;
 and A:  1B;
 OK:  ;
EDIT C:  Done OK: _;
```

Good. Now print branch 1 to observe the result.

TASK 67

```
EDIT C:  Print OK:/C:  C:  Branch at A:  1;
 V: _;
 1  JUICE
    1A  GRAPEFRUIT
    1B  ORANGE
EDIT C:  Done OK: _;
```

To make a larger change, please transpose branches 2 and 4.

TASK 68

```
EDIT C:  Transpose C:  Branch at A:  2;
 and A:  4;
 OK: _;
EDIT C:  Done OK: _;
```

Please check your result by printing the entire file.

TASK 69

```
EDIT C:  Print OK:/C:  C:  Branch at A:  0;
 V: _;
  <NLS-SCHOLAR>BREAKFAST.LNLS;1  01-OCT-74  08:20  CLH  ;
   1  JUICE
      1A  GRAPEFRUIT
      1B  ORANGE
   2  BEVERAGE
      2A  HOT CHOCOLATE
      2B  TEA
          2B1  WITH LEMON
          2B2  WITH SUGAR AND CREAM
      2C  COFFEE
   3  EGGS
      3A  FRIED
          3A1  SUNNY-SIDE-UP
          3A2  EASY-OVER
      3B  BOILED
      3C  SCRAMBLED
   4  CEREAL
      4A  OATMEAL
          4A1  WITH RAISINS
      4B  CREAM OF WHEAT
      4C  CORN FLAKES
EDIT C:  Done OK: _;
```

You can see that the beverages are now in branch 2 while the cereals are in branch 4. The branch "JUICE" and the branch "EGGS" are still numbered as before.

There is no restriction that the statements or branches which are tranposed be at the same level, but for many applications this will be the case.

QUESTIONS?

**

OTHER STRUCTURE UNITS

So far the only structure units we've manipulated have been statements and branches. There are two other kinds of structure units in NLS -- plexes and groups. Since both are defined in terms of branches, let's be sure that the concept of branch is well understood.

A branch consists of all statements whose statement numbers begin with the same fields as those specified. Thus branch 2B2 consists of all statements whose statement numbers begin with 2B2. If there are no other statements whose numbers begin that way, then branch 2B2 consists only of statement 2B2.

Another way of describing a branch is to say that it consists of a specified statement, and all its substatements, and all their substatements, etc. You can see from this definition why branch 0 consists of all statements in a file.

In order to understand what a plex is, the term 'source' must be defined. Source is the inverse of substatement. The source of a statement is always one level higher than the statement itself. The source statement is obtained by removing the last field from the specified statement number. Thus the source of statement 2B2 is statement 2B; the source of statement 1A is statement 1. The source of all first-level statements (1,2,3,4) is statement zero. Since there is no statement which is one level higher than statement zero, it has no source.

QUESTIONS?

**

PLEXES

A plex is a set of branches, much as a branch is a set of statements. A plex consists of the specified branch together with all other branches which have the same source. Thus plex 3A consists of branches 3A, 3B, and 3C in our BREAKFAST file; that is, all the kinds of eggs. Plex 3A, plex 3B, and plex 3C are all designations for exactly the same thing.

In the most recent BREAKFAST file, what statements are in plex 2A?

2A 2B 2B1 2B2 2C

THAT'S FINE.

A plex is a useful concept since it allows you to manipulate a set of statements all below a certain level. Use the term plex to delete all the particular kinds of cereals, while leaving the heading CEREAL intact.

TASK 70

EDIT C: Delete C: Plex at A: 4A;
OK: _;
EDIT C: Done OK: _;
Please print the branch "CEREAL" to view your results.

TASK 71

```
EDIT C:  Print OK:/C:  C:  Branch at A:  4;
 V: _;
 4  CEREAL
EDIT C:  Done OK: _;
```

Note that deleting plex 1 (or 2 or 3 or 4) would delete everything in the file except statement zero.

QUESTIONS?

**

GROUPS

A group consists of a set of contiguous branches of a plex. To define a group, two addresses must be given: the first branch and the last branch. The group then consists of the first branch and the last branch and all other branches, if any, that lie in between. Thus the group 2 through 3 consists of all the statements in branches 2 and 3; the group 4A through 4C consists of statements 4A, 4A1, 4B, and 4C.

What statements are in group 3A through 3C?

3A 3A1 3A2 3B 3C

YOU ARE CORRECT.

Note that the two addresses given to specify a group must have the same source. If this is not the case, the message ILLECAL GROUP will be printed.

Use the term group to delete all eggs except scrambled eggs from the breakfast menu

TASK 72

```
EDIT C:  Delete C:  Group at A:  3A;
 through A:  3B;
 OK: _;
EDIT C:  Done OK: _;
```

Good. Please print the branch "EGGS" to see your results.

TASK 73

```
EDIT C:  Print C:  Branch at A:  3;
 V: _;
 3  EGGS
    3A  SCRAMBLED
EDIT C:  Done OK: _;
```

QUESTIONS?

**

Try using either 'group' or 'plex' to incorporate the juices into the branch "BEVERAGE". That is, move the statement containing "GRAPEFRUIT" and the statement "ORANGE" so that they are the first choices of the BEVERAGES. Then delete the statement "JUICE".

TASK 74

```
EDIT C:  Move C:  Group from A:  1A;
 through A:  1B;
 to follow A:  2;
 L:  d;
EDIT C:  Delete C:  Statement at A:  1;
 OK: _;
EDIT C:  Done OK: _;
```

Test your result by printing the branch "BEVERAGE" of the file.

TASK 75

```
EDIT C:  Jump to  A:  0;
EDIT C:  Print OK:C:  C:  Branch at A:  "BEVERAGE"
 V: _;
 1  BEVERAGE
    1A  GRAPEFRUIT
    1B  ORANGE
    1C  HOT CHOCOLATE
    1D  TEA
        1D1  WITH LEMON
        1D2  WITH SUGAR AND CREAM
    1E  COFFEE
EDIT C:  Done OK: _;
```

QUESTIONS?

**

REVIEW OF LESSON THREE

Lesson Three has introduced the following commands:

Print command for structure units
Set Viewspecs command
Show Viewspecs command
Move command
Copy command
Transpose command

QUESTIONS?

**

It has introduced the following viewspecs:

a show one level less
b show one level more

m turn statement numbers on
n turn statement numbers off

w show all lines, all levels
x show one line, one level

y turn blank lines on
z turn blank lines off

A turn indenting on
B turn indenting off

QUESTIONS?

**

It has introduced the following concepts:

viewspecs
standard viewspecs
V: prompt

shift character
illegal move

source
substatement
group
plex

QUESTIONS?

**

This introduction to the EDIT subsystem of NLS is now complete. Please feel free now to use the system as much as you like and to ask any questions which may arise. Type 'Done <CR>' when you're finished, as usual.

TASK 76

EDIT C: Done OK: _;

I've enjoyed talking to you. Goodbye.