DOCUMENT RESUME

ED 163 972 IR 006 645

AUTHOR	Hinckley, Michael; And Others
TITLE	VOCAL: Voice Oriented Curriculum Author Language. Technical Report No. 291.
INSTITUTION	Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	19 Oct 77
GRANT	EPP-76-15016-A01; SED-74-15016-A02
NOTE	47p.; For related documents, see IR 006 644-647
EDRS PRICE	MF-$0.83 HC-$2.06 Plus Postage.
DESCRIPTORS	*Auditory Perception; *Computer Assisted Instruction; Computer Programs; *Curriculum Development; Data Processing; *Display Systems; *Programing Languages
IDENTIFIERS	Computer Software; *VOCAL

ABSTRACT

VOCAL (Voice Oriented Curriculum Author Language) is designed to facilitate the authoring of computer assisted curricula which incorporate highly interactive audio and text presentations. Lessons written in VOCAL are intended to be patterned after the style of informal classroom lectures. VOCAL contains features that allow the author to specify audio messages in several formats acceptable to the audio hardware and associated software, and to control the interaction of the audio presentation with material presented visually on the screen of a CRT terminal. This description of VOCAL includes an elaboration of its features, the Browse Mode, the Help System, and operation of the VOCAL Compiler and Interpreter. Appendices include a file format for compiled VOCAL files and implementations of the S Opcode. (CMV)

VOCAL: VOICE ORIENTED CURRICULUM AUTHOR LANGUAGE

by

Michael Hinckley, Robert L. Laddaga, John Prebus, Robert Smith,

and David Ferris

Technical Report No. 291

October 19, 1977

Reproduction in Whole or in Part is Permitted for
Any Purpose of the United States Government

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

Stanford University

Stanford, California 94305

Acknowledgments

The authors wish to thank Patrick Suppes for his able direction of this project. Robert Smith, Michael Hinckley, and Robert Laddaga did the majority of the design work. The original implementation was done by John Prebus with assistance from Robert Smith. David Ferris designed the help system, wrote HELPMK, and assisted John Prebus in writing the lesson driver code needed for the help system. Extensive rewriting and modularizing of the original programs was carried out by Larry Markosian and Scott Daniels. Teri Pettit has rewritten the help system programs and assisted in revising the sections documenting the help system. She, along with Mark Davis, made many important suggestions for the features implemented in VOCAL. The authors also wish to thank Dianne Kanerva for her excellent editorial assistance. The work described here was supported under NSF Grants EPP-74-15016 A01 and SED-74-15016 A02.

Table of Contents

Section Page

Subsection

1 Introduction

VOCAL (Voice Oriented Curriculum Author Language) is designed to facilitate the authoring of curriculums which incorporate highly interactive audio and text presentation. Lessons written in VOCAL are intended to be patterned after the style of informal classroom lectures. VOCAL contains features that allow the author to specify audio messages in several formats acceptable to the audio hardware and associated software, and to control the interaction of the audio presentation with material presented visually on the screen of a CRT terminal.

There are several reasons for incorporating audio messages in a computer-assisted instruction course. One reason is to more actively involve the student in the instructional process by using his sense of hearing in addition to his sense of sight. A second reason is the obvious advantage of having a second channel for communicating with the student. A third reason is that certain sorts of information can most appropriately be presented to the student via an audio channel. For example, it is often useful to make informal comments about formulas or tables which are simultaneously being displayed. Finally, certain kinds of semantic information are usually carried by complex emphasis patterns in speech. Such information can be difficult or awkward to reproduce in writing.

Given a system for generating and storing audio messages for random access by a program providing a course of study to students, curriculum authors need a systematic way of indicating what material is to be presented by the audio system, and what material is to be presented by display or typescript. Authors also need ways of indicating complex information about how the audio and visual modes of presentation are to be coordinated. Such facilities are provided by VOCAL, as well as more traditional facilities such as complex question and answer procedures. VOCAL has also been designed to facilitate the addition of course-specific exercises such as proving theorems in a logic course.

The syntactic structure of the language is quite simple and compressed and is based on LISP-style S-expressions. We selected this syntax because it is easily parsed and compiled, and the resulting language retains a flexibility and extensibility that a more algebraic language might lack. The template features of VOCAL are inspired by features of the PUB document preparation language (Tesler, 1972); we share with PUB the belief that computers can do more text processing than they are generally permitted. The SAIL programming language (in which VOCAL is embedded) has contributed some features. The functions for question/answer analysis were suggested by such CAI languages as INST and COURSEWRITER.

While VOCAL is intended to be a general system for preparation of CAI materials utilizing audio, we have included a number of special functions that relate to the particular use of VOCAL in writing curriculum for Stanford's logic and other mathematically based courses.

Only limited programming features are included in VOCAL. For example, only certain "conditional" operations have been programmed, and no explicit data declarations are available. VOCAL is not a general programming language. Instead, it is a language for tightly expressing particular interactions between the audio output, the visual output, and the student. Our implementation emphasizes the interface between programming, which is done in SAIL or LISP, and authoring, which is done in VOCAL.

A classical problem in designing an author language is that, on the one hand, one wants a language that authors can use with minimal training in programming, a language in which presentation of materials and response judging will be straightforward; on the other hand, it is often necessary to have a genuine programming language, with sophisticated data and control structures, for application in CAI environments that emphasize problem solving, dialogue, and the like. Some CAI languages solve this problem by restricting what is possible or convenient to do (INST, COURSEWRITER); other languages maintain the claim to universality, but do so with data and control structures that are quite primitive (TUTOR). The current popularity of BASIC as an author language can be traced to the fact that it is quite simple and yet is a complete programming language; it is also widely available.

We think that a better solution to the problem of author languages is to select the best programming languages available for writing the program portions of the CAI system and to interface the resulting programs to special-purpose author languages for the curriculum material. VOCAL is such a special-purpose language.

This report is intended primarily as a description of the VOCAL language and its features. Readers who are interested in the pedagogical applications of the language or who would like a tutorial introduction to the use of VOCAL should see (Davis and Pettit, forthcoming), a useful supplement to the material in this document.

2 The VOCAL Language

This section describes the syntax and semantics of the VOCAL language. A short overview of the various operation codes (opcodes) described below may help in clarifying their uses.

The LESSON, EXERCISE, BEGIN and HOLD opcodes serve the purpose of organizing potentially vast amounts of curriculum material for both the curriculum author and the student. The display opcodes (such as TEM, T, B, etc.) allow the author to specify what material will appear on the display terminal screen and its format. In addition, the author can seperately specify the order in which the material is displayed.

The audio opcode, S, specifies what material is to be spoken. When an opcode which controls the display on the screen is followed by a text string (both within the scope of the same S opcode), the audio system will begin speaking the text at the same time that the display system begins to execute the display opcode. This allows the curriculum author to syncronize the audio and display systems. There are also timing opcodes which give the author additional control.

The question-related opcodes are used to present students with short answer questions and to analyze the student's input. The author is allowed to specify the number of chances the student has to answer, the program's response to correct or incorrect answers, whether to respond differently to specific correct or incorrect answers, and whether to ask further questions of students who fail to correctly answer a question. Facilities are provided for prompting the student as to time and manner of required response. In most cases, the author can accept reasonable defaults in the question and answer procedure if he does not desire to use all of the available features.

The conditional opcodes AUDIO, NONAUDIO, DPY, and TTY, are used to indicate alternative code to be executed depending on the type of terminal at which the student is working. The concern is whether the terminal allows sufficient control of the display for the DPY and AUDIO versions, and whether the terminal is provided with the appropriate audio equipment.

The branching opcodes allow the author to control the progress of the student through the curriculum. At a minimum the author should specify a linear move from lesson to lesson by including a GOTO at the end of each lesson, pointing to the next lesson. More complicated paths can be arranged by the author, where final choice of path for an individual student may depend upon his own choice or upon his meeting some preset criterion.

2.1 Conventions

The basic syntax of VOCAL is similar to LISP 1.6. The S-expression is used to code the instructions. The CAR (first element of the list) is the name of the operation code ("opcode"). Just as in LISP, the CAR of a list is treated as the name of a function. Opcodes may embed within each other. Indeed, it is the power of embedding, so easily achieved with the LISP syntax, that has led us to the choice of a very LISP-like syntax.

For example, the following VOCAL statement defines the body of an exercise (number "1") in a lesson:

```
[EXERCISE 1 "Sample exercise"
(AUDIO
         (T "This is the text of the exercise.")
         (S "Notice the text of the exercise on the screen."))
(NONAUDIO
         (T "This is the text of the exercise."))]
```

The S and T opcodes, respectively, speak and type their arguments. The AUDIO opcode says that the statements contained within its scope are to be done if the student is presently using the audio system. The statements within the scope of the NONAUDIO opcode will be done if the student is not presently using the audio system. The entire text of the exercise is surrounded by the EXERCISE opcode.

Like some versions of LISP, the VOCAL language allows the use of square-brackets in place of parentheses. In addition to closing the most recent left square-bracket, a right square-bracket will close any open left parentheses within the scope of the square-bracket pair. Judicious use of square-brackets will enhance the readability of the lesson source file and serve as a debugging aid by limiting the scope of a parenthesis mismatch error.

In the sequel several conventions will be used to state the details of the language. The term "string" will refer to a literal string constant, i.e., a piece of text surrounded by quotation marks. Tokens in upper case are VOCAL opcode names. Tokens in lower case, surrounded by "< >", are metalinguistic.

2.2 LESSON Opcode

The lesson opcode is syntactically of the form:

(LESSON <number> "description" <actions>)

where:

<number> is the lesson number.

"description" is a single line description of the lesson.

<actions> is an arbitrary sequence of VOCAL actions to be performed if the student, in "Browse Mode" (see Section 3), asks for information about this lesson.

The LESSON opcode should occur only once in a given VOCAL file.

2.3 EXERCISE Opcode

The EXERCISE opcode has the syntactic form:

(EXERCISE <number> "description" <actions>)

where:

<number> is the number of this exercise.

"description" is a one-line description of this exercise, to appear at the top of the display screen.

<actions> is the sequence of VOCAL actions for this exercise.

An EXERCISE is an entry into the curriculum, the smallest unit that can be randomly accessed. VOCAL files should therefore have the following form:

```
[LESSON n "lesson description string" <actions>]
[EXERCISE 1 "..." <actions>]
[EXERCISE 2 "..." <actions>]
    ........................
[EXERCISE n "..." <actions>]
```

Only compiler-specific opcodes should occur outside the scope of a LESSON or EXERCISE opcode. If there is another lesson to follow, the GOTO opcode should be the last action of the last exercise.

2.4 BEGIN Opcode

The BEGIN opcode groups together a number of statements that are to be executed as a single unit. For example,

```
(BEGIN  (T "Now is the time for all good men")
        (S "Now is the time for all good men"))
```

both types and speaks "Now is the time for all good men". Note that a list of lists, e.g. ((T "....") (S "....")), begins with an implicit BEGIN.

2.5 Display Opcodes

The display screen is divided into four portions. The first portion is a single line, called the Description Line. This line is reserved for the "description" string of an EXERCISE opcode or its analogue in Help or Browse Modes. One of its major functions is to help the student keep track of where he is. The second portion is called the Template Region. It gets its text from the template string as described below. The third region is a single line, called the Mark Line. It serves as a visual delimiter between the Template Region and the fourth region, which is called the Scroll Region. Some opcodes place text in the Scroll Region, but the text is always added to the bottom line of the screen, and old text moves upward, to disappear when it reaches the Mark Line.[1]

[1] The screen is manipulated using a device-independent display package developed by the systems staff at IMSSS. The advantage of this approach is that neither the lesson author nor the program author needs to be concerned with producing different code for the various kinds of display terminals in use at the Institute.

The syntax for setting up a template region is discussed in the section on template opcodes; the syntax for controlling run-time display of text in the template and scroll regions is discussed in the section on display-control opcodes.

2.5.1 Template Opcodes

The TEM and TEM2 opcodes set up a template, described below, which contains the displayed text. The syntactic form of TEM is:

(TEM "template string" <actions>)

TEM2 is an extension, which is described below.

The "template string" is an image of the display screen. This image contains area pointers that point to logical lines, groups of lines, and contiguous segments of lines.

The token "<actions>" indicates a sequence of statements that are executed. Any opcodes for display control (see below) may refer to the logical areas of the template. When the template is entered, the screen is cleared (except for line 0), and the scroll point is set to the line that is the bottom of the template. A TEM opcode cannot contain another TEM opcode; in other words, the templates do not embed.

The "template string" is a string constant that allows one to print, overprint, erase, or brighten[2] any part of the template.

(TEM2 "template string" "additional string" <actions>)

This is like TEM except that the 'additional string" may contain areas and subareas that are to be overtyped by the OT opcode or its relatives. The template string still defines the shape of the template. Details and restrictions are indicated below.

2.5.2 Areas and Subareas Within a Template

The template string serves several purposes simultaneously. First, the total number of text lines, including empty lines, determines the size of the template region. This is currently limited to 19 lines of text since the Datamedia terminals, which hold a maximum of 24 lines, are the limiting case.[3] At least three lines are needed for the Scroll

[2] How the text is brightened depends on the hardware. On some terminals the text is printed in double-intensity mode, hence the term 'brightened'. On other terminals the text will be underlined and on still others the text will blink or flash.

[3] It is usually a good idea to leave the first line of text blank to serve as a visual delimiter between the Description Line and the text, so in practice there are 18 lines available for text. In exceptional cases the limit can be extended to 21 lines, leaving only one line for the Scroll Region. But this is rarely a good idea.

Region, and the Description and Mark Lines each take one. Second, the text which will appear in the Template Region is introduced. And finally, associations are made between the text and parts of the template region. There are two ways of explicitly designating parts of the template. One way is to give a name to a line or group of contiguous lines, which is then called an <u>area</u>, and is specified by using the special character "%".[4] The other is to designate a contiguous part of a single line as a subarea.

Consider the following template string:

```
"


      (1)  P                                        %1_
           A
      (2)  P -> Q                                   %2_
           B CC D


(2) means that                                      %
      if P is true                                  %_
      EEEEEEEEEEEE
              then Q is true                        %3


"
```

In the above sample template string, the lines that contain "%" are designated as the beginning, continuation, or end of an area spanning one or more complete lines. Lines that end with "%" mark the first line of an area that is to extend over two or more lines, with the next line that ends with "%<number>" being the final line of that area. Note that once a multiline area is begun with a line ending with "%" (or with "%_"), the area is terminated only by a line ending with "%<number>". There may be intervening lines ending with "%_" that are simply continuation lines.

The most complex cases are the lines that end with either "%_" or "%<number>_". The first of these says that the next line is not actually to appear on the screen, but instead will contain subarea designators into the current line. Thus, "if P is true" is designated as a subarea (named "E"), consisting of a number of column positions. Likewise, a line that ends with "%<number>_" says that <number> marks the current line(s) as an area with name <number> but that, in addition, parts of the current line will be marked by the area designators on the next line.[5]

[4] This character is dynamically resettable using the TEMCHAR opcode, q.v.

[5] The "_" symbol is special only when it occurs to the right of a "%". Otherwise it is treated as ordinary text.

Note: Area and subarea designators never appear on the screen. The compiler uses them to give names to areas and subareas and then discards them. For example, if an author requested that areas 1, 2, and 3 of the sample template string above be displayed on the screen, the student would see the following:

```
(1)  P
(2)  P -> Q
```

(2) means that
    if P is true
        then Q is true

<number> can be any positive integer, e.g, 7 or 32. A subarea designator can be any alphabetic character. BUT CASE MAKES A DIFFERENCE! In other words, the compiler will distinguish between "a" and "A". This gives a total of 52 distinct subarea designators.[6] A subarea designator can be used only once in a template string and all occurrences must be contiguous. So "AAA b C" and "AAC b" are legal, but "A A b" and "AbA" are not.

Several opcodes--T, E, U, OT, OB, OE--may refer to areas of the current template by being included in the <actions> list within the scope of the template. This is explained below.

2.5.3 Display Control Opcodes

The following opcodes generate the display on the screen. The metalinguistic "<areas>" means any sequence of area or subarea designators that occur in the scope of a TEM or TEM2 opcode. It is an error to have an area reference that does not occur within the scope of TEM or TEM2.

(T <areas>)

Each area in <areas> is typed out in the appropriate region of the template. <areas> can be replaced by a string constant which will be typed out in the Scroll Region.

(E <areas>)

Each area in <areas> is Erased, leaving those areas blank on the screen, but leaving the remainder of the screen undisturbed. The action (E) with no arguments erases the entire template (but not the Scroll Region).

(B <areas>)

[6] If more than 52 subarea designators are needed, the digits are also legal subarea designators provided they are not used as area designators. This could be accomplished by using two-digit area designators.

Each area in <areas> is "Brightened"--typed out in double intensity mode, blink mode, or underline mode depending on the physical device available and its characteristics.

(U <areas>)

Each area in <areas> is "Unbrightened"--leaving the text typed out. (U) with no arguments unbrightens all brightened areas. Note: there is no implicit unbrightening, so if an author wishes to restore an area to normal state, he must expressly unbrighten that area. Unbrightening restores the text; to erase use the E opcode. If an area has been erased, unbrightening it causes it to be retyped in normal mode.

(M)

The M (for "Mark screen") opcode prints a dotted line on the Mark Line of the display region in a TEM or TEM2 opcode. A dotted line is inserted there automatically by the first Q or CQ opcode executed within the TEM or TEM2, so when these opcodes occur within the scope of a TEM or TEM2, the M opcode merely allows the author to insert the dotted line sooner if he wishes.

(OT <oldarea> <newarea>)

OverType <oldarea> with <newarea>. The text from <newarea> is picked up and inserted into <oldarea>. <oldarea> must be a single area in the "template string" of a TEM2 opcode and <newarea> must be either an area in the "additional string" of a TEM2 opcode, or a string constant that is just printed in <oldarea>. If <newarea> and <oldarea> are not the same size, the text of <newarea> will be adjusted to fit <oldarea> by truncation or filling in with blanks or blank lines as required. It is better for the author to ensure that <oldarea> and <newarea> occupy the same number of lines; the result will then be obvious.

(OE <oldarea> <newarea>)

OverErase <oldarea> with <newarea>. This is the same as OT, only the <newarea> is not displayed; <oldarea> is erased, and <newarea> and areas sharing screen locations with it (see below) become defined.

(OB <oldarea> <newarea>)

OverBrighten <oldarea> with <newarea>. The same as OT, only the <newarea> is displayed in brighten mode.

2.5.4 Remarks on the Implications of Overtyping

In the simplest case, an "area" or "subarea" is defined to be a region on the display screen plus the text that is currently displayed there. However, in the case of a subarea, this is not quite true--since the text of such a subarea is shared with the text of an area containing the entire line. When using the TEM2 opcode it is important to keep in mind that it is the first template string that defines the basic size and shape of the Display Region. But the conventions are somewhat different for subareas. We illustrate the conventions by example.

```
(TEM2 "
          Now is the time.              %1_
          AAAAAA
      "
      "
          Here's a good example.        %2_
          BBBBBB       CCCCCCC
      ")
```

If the opcode (OT 1 2) is executed, the VOCAL convention is that area 2 becomes actively defined in the template region. For example, (B 1) and (B 2) will both cause "Here's a good example." to be brightened on line 1. But the convention for subareas is different. When (OT 1 2) or (OB 1 2) or (OE 1 2) is executed, the subareas of area 1 are no longer active. Instead, the subareas of area 2 become activated. So (B A) is no longer legal at that point.

It is clear that some convention is necessary; otherwise, after the opcode (OT 1 2) in the example, the opcode (B A) would be ambiguous. The convention chosen will generally work to the lesson author's advantage by simplifying the task of brightening small areas after they have been displayed on the screen as a result of overtyping.

The complier will not complain if a subarea instead of an area is used within the scope of an OT, OB, or OE opcode, but the results may well not be what was intuitively expected. In fact, one of the major uses of the OE opcode is not only to allow an "old" template area to be erased, but also to activate the subareas in the new text string, so that a call such as (T B) will have the desired effect.

2.6 Audio Opcodes

The S opcode is interpreted by calls to functions in the audio library, written by W. R. Sanders and G. Benbassat.[7]

```
(S <display!action> "message" <display!action> "message" ... )
```

This is the general form of the Speak opcode. Each <display!action> is either a display opcode or a list of display opcodes (all other opcodes, except W, are illegal), e.g., (B 4 A) or ((T 4) (B E) (T F)). A <display!action> is executed simultaneously with the speaking of the following "message". The opcode is implemented so that the "messages" are spoken continuously without interruption; that is, the "messages" are treated as a single paragraph. Thus the lesson author can synchronize the screen display with a specific word in the sentence. VOCAL will wait until all speaking and other actions are completed before beginning the next opcode. All of the <display!actions> and all but one of the "messages" may be omitted. For example:

[7] See Sanders et al., 1976 for an overview of the audio system.

```
(S "Now is the time" (T "Buy bonds") "for all good men")
```

will display "Buy bonds" just as "for all good men" is being spoken. Caveat: Only the start times of the <display!action>s and spoken "messages" is coordinated. Thus, if the execution time for <display!action> exceeds that of the following "message", later synchronization may be lost. Under heavy system load, execution times for the <display!action>s will of course increase. The author should take note of this when deciding how many "messages" and how much typing to include within the scope of a single S opcode.

Exactly how the "messages" will be spoken depends on the current implementation.[8] Briefly, messages may be prerecorded and processed for storage in a ‘compressed’ form, or the messages may be ‘constructed’ at compile time from prerecorded words or morphemes, with syntactic prosodic features added by the compiler. The former method is referred to as ‘long sounds’, the latter as ‘prosody’.

At least two audio ‘languages’ are available to lesson authors. The language for long sounds is the compressed, digitized representation of the recorded phrases. A language for prosody will generally be a dictionary of recorded words, also in compressed, digitized form. The language opcode allows the author to change the language or mode of audio presentation within a lesson.

```
(LANGUAGE "audio language name" "long sounds flag" "prosody flag")
```

If the string "long sounds flag" is set to "TRUE", then the student hears long sounds. If "long sounds flag" is set to "FALSE" and "prosody flag" is set to "TRUE", then the student hears prosody. The author must select the proper language in each case. If the "long sounds flag" and "prosody flag" are both set to "" (NIL), then the mode of audio presentation will remain unchanged, allowing a change of language only. It is possible to change languages without changing the mode, as long as the unit of recording (word or phrase) is the same for both languages. It is NOT, in general, possible to change modes without changing the language, since the required unit of recording will be different for the different modes. Thus, if the author wants to shift from long sounds mode to prosody mode, he must also switch from a language of stored phrases to a language of stored words.

In an effort to include special (semantic) emphasis in the ‘prosody’ version (and, in addition, aid a human recorder), emphasis markers are available to lesson authors. $<num> (followed by a space) may be inserted before the word to be emphasized in the speak text. <num> may be either 1, 2, or 3, where 1 indicates the greatest stress and 3 the least. All three give an increase over the normal, syntactically generated stress. For example:

```
(S "$1 Now is the time for all $3 good men")
```

will put a great deal of stress on the word ‘now’ and just enough stress on ‘good’ to cause a listener to pay some extra attention to the word.

[8] See Appendix B for details.

2.7 Timing Opcode

Most of the timing and synchronization is handled automatically by the compiler/driver (see the "S" opcode above). There are a few cases where authors will wish to specify a short break in the presentation, such as at the end of a logical "paragraph" of the lecture. This is done by the W opcode.

```
(W <mstime>)
```

<mstime> is the number of milliseconds for the dismiss. If <mstime> is missing, a default value is used, determined by the compiler.

The W opcode is the only nondisplay class opcode that is legal inside the scope of an S opcode. There, instead of generating a dismiss, it causes the audio hardware to generate silence for the indicated number of milliseconds. At times of higher system load, W's outside the scope of S opcodes will produce pauses of greater length than their arguments, while W's inside the scope of an S opcode will not.

Inside a speak, the W opcode functions exactly as a message string. Preceding display actions are executed just as the silence begins its execution; subsequent display actions are executed only after the specified time of silence. The syntax for the W opcode outside of an S opcode is the same as that of any action. Inside the scope of an S opcode, the syntax is the following: A W to be executed with a series of actions should be included as the last of the actions in an implicit BEGIN. Actions that are intended to be executed after a W should be enclosed in a separate implicit BEGIN. Also, a W opcode cannot be immediately followed by another W (this is of course no semantic restriction, since the same effect is achieved by adding the arguments).

```
(TEM "
     Pick the correct answer                          %1_
              AAAAAAA                                   "

(S "Pick the correct answer,"
   ((T 1) (W 1000)) ((B A) (W 500))
   "to the next question."))
```

The effect of the above code would be that first the phrase `pick the correct answer' would be spoken, second the phrase would begin to be typed, and simultaneously there would begin one second of audio silence. The word `correct' would be brightened immediately after either the second of silence, or the typing of the phrase, depending on which takes longer. In either case one half-second of silence would begin simultaneously with the brightening action, and at the completion of both the audio silence and display actions, the phrase `to the next question' would be spoken.

2.8 HOLD Opcode

There are numerous opcodes in VOCAL that ask for a response from the student. The simplest of these allows the student to state when he is ready to continue. This is the HOLD opcode.

```
(HOLD "prompt string" <actions>)
```

In its simplest form, (HOLD), the HOLD opcode simply types out

```
[Type ESC to go on.]
```

and then waits for the student to type something before continuing. The purpose of this opcode is to let the student judge when he is finished examining the display. No answer analysis is done; however, functions like logging out, Browse, Help, and repetition of text all work from this opcode, since they are embedded in a lower-level function in the interpreter.

The HOLD opcode may also take arguments. If the first argument to HOLD is a string constant, then that string is embedded in the prompt string:

```
"[Type ESC to go on, CTRL-A to " + "prompt string" + ".]"
```

So, if "prompt string" were "repeat the argument", the result would be

```
"[Type ESC to go on, CTRL-A to repeat the argument.]"
```

If <actions> is nonempty, but "prompt string" is omitted, the default prompt string is "repeat". <Actions> is a sequence of opcodes to be executed before the HOLD is done (and to be repeated on receipt of CTRL-A). It is recommended that small reentrant blocks of code, i.e., blocks of code that can be re-executed without obtaining a poor effect on the screen, be surrounded by HOLD in order to give the student a chance to repeat should he not fully understand them the first time.

2.9 The Q and CQ Opcodes

The rest of the response-obtaining opcodes are the most complex in the VOCAL language, because a number of independent features may occur within their scope. The first of these, the Q opcode, is the basic answer command. The syntax of the Q opcode is:

```
(Q <tag> <action> <tag> <action> . . . )
```

As an example of the operation of Q, suppose we have the following:

```
(Q    INIT    ((S "What is the first letter of the alphabet?")
               (T "Type the letter indicating your answer."))
      A       (ANS "A")
      HINTL   ((S "The letter before B") (S "Think harder"))
```

```
     CA       (S "Good")
     WA       (S "No, the answer is A"))
```

In the above, the tags A, CA, WA, and HINT are used to label the action component that follows the tag. The example is interpreted as follows: First, the sentence "What is the first letter of the alphabet?" is spoken, and will be repeated each time the student types CTRL-A. Next, the sentence "Type the letter indicating your answer" is printed on the terminal, followed by a second line beginning with "*". The program will then pause until the student types something. If the student types CTRL-H he is told "The letter before B"; if he types CTRL-H again (any number of times) he is told "Think harder" (each time). If he types the letter "A" he is told "Good" and the exercise is ended. If he types anything else, he is told "No, the answer is A", and the exercise is ended, since no R tag is given (i.e., no repeats are specified--see Section 2.9.1).

The CQ (Cascading Questions) opcode makes it easy to code exercises that ask one question, then ask another if the first is wrong, then another, etc. The components of each list in the CQ opcode can be any of the tags described in Section 2.9.1.

```
(CQ (<list of tags and actions>)
    (<list of tags and actions>)
    ...)
```

Each <list of tags and actions> is a separate question and has exactly the same format as described for the Q opcode with the sole difference that the Q is omitted at the beginning of the list. Questions are given until one is correct, or the list of exercises is exhausted.

2.9.1 Tags for the Q and CQ Opcodes

The following tags are available for the answer-obtaining opcodes:

1. INIT (<againaction> <promptaction>). <Againaction> is executed upon first entering the exercise (and repeated for CTRL-A until a response is matched). It is either a single action or a list of actions. <Promptaction> is executed whenever input from the student is requested. The defaults for these actions are to do nothing. It is recommended that the full discussion of the exercise at hand be included in the <againaction> and a brief description of the type of input requested be included in the <promptaction>, e.g., (T "Answer Y or N").

2. A <answeranalysis opcode>. Analyzes the answer. The answer-analysis code following this tag specifies the correct answer.

3. CA <action>. The action to perform if the answer is correct; this also finishes the Q opcode. The

default is to say that the student's answer is correct, using a randomly selected correct-answer response. In other words, if the author does not specify an action, the program will answer with "good" or "excellent" or some other appropriate response. This response is selected at random from a list stored by the program.

4. CAS (<correctanalysis> <action> ...) (Correct Answer Select). This is a list of pairs of answer-analysis opcodes and actions with the interpretation that the first of the <correctanalysis> opcodes that causes a match will cause the corresponding <action> to be performed; the exercise will be considered correct.

5. WA <action>. Action to take if the answer is wrong; this finishes the opcode if no repeats were specified. The default is to say that the answer is wrong using a randomly selected response.

6. WAL (<action> <action> ...). This is a list of actions to be given one at a time, each time the student gives a wrong response, until the last item in the list. The last <action> in the list will then be treated just like the <action> following a simple WA. That is, the question will be furthur repeated the number of times specified by the "R" opcode, or until a correct answer is given. Each wrong answer will cause the last <action> in the list to be given until the last repeat of the question when the FAIL actions will be executed, if they have been coded.

7. WAS (<wronganalysis> <action> ...) (Wrong Answer Select). This is a list of pairs of answer-analysis opcodes and actions with the interpretation that the first of the <wronganalysis> opcodes that causes a match will cause the corresponding <action> to be performed. In any case, the exercise is considered wrong, since it is assumed that any CA or CAS has already been executed. The purpose of this tag is to catch specific wrong answers.

8. R n. Here, "n" is the number of times to ask for an answer, and is defaulted to 1. For example, R 2 would mean to ask the question, and then repeat it once if it is answered wrong the first time.

9. FAIL <action>. This specifies the action to take at the end when the student has failed, in R tries, to answer the question correctly. It is useful for exercises that contain an R, since FAIL can be used to inform the student of the correct answer. It is an error to use the FAIL opcode when only one attempt is allowed. In other words, when using FAIL there should also be an R n for some n greater than 1.

10. HINT <action>. This is the hint to give the student if he asks for one. The default is to say that no hint is available.

11. HINTL (<action> <action> ...). This is a list of hints to be given, one each time CTRL-H is typed; the last <action> in the list will be repeated as often as requested.

12. COMMENT (<anything>). Whatever is within the scope of the immediately following set of parantheses is ignored, though Quote marks and parentheses must be properly paired so that VOCAL can detect the end of the comment. Note that in this construction, COMMENT is used as a TAG, not as the CAR of a list.

These tags and their corresponding actions may occur in any order. Either an A or a CAS tag <u>must</u> occur; but all other tags are optional. It is an error for a tag (except COMMENT) to occur more than once in an answer-fetching opcode. Some of the combinations are illegal. For example CA and CAS should not occur within the scope of the same Q opcode. Also, at most one of the wrong answer tags (WA, WAL, WAS) may be included in the scope of a Q opcode.

2.9.2 Answer Analysis Opcodes

Once a response (answer) is obtained from the student by the use of the Q or CQ opcodes, it must be checked. This is done by the answer-analysis opcodes. The analysis is done by testing whether there is a match between what the student has typed and the coded answer(s).

For example, the command:

```
(Q A (ANS "74"))
```

gets a response from the student, and then analyzes it by comparing for exact string equality with "74" for correctness. (In this example, the other parts of the Q opcode are omitted, thus defaulted.) Except for TRANS, all of these opcodes are quite simple and standard.

Several kinds of editing are done by these opcodes on the responses that the student types. All of the opcodes remove beginning and trailing spaces and tabs, and all of them compress multiple spaces and tabs to a single space. These procedures are called <u>space compaction</u>. Additionally, most opcodes convert lower to upper case, called <u>upper conversion</u>. A few opcodes that are prepared to accept several answers will scan for those answers separated by spaces or commas. This is called <u>word scanning</u>.

The student's input is referred to as <response> below.

(ANS <sequence of strings>)

After space compaction and upper conversion of <response>, ANS checks to see if any string in <sequence of strings> is exactly the same as <response>. In other words, the student must type exactly one item from the sequence for a match to occur.

(ALLANS <sequence of strings>)

After space compaction, upper conversion, and word scanning of <response>, ALLANS makes certain that all of the strings in <sequence of strings> were typed by the student, and that no additional strings were typed. So the student will have to type in all, and only, the items in the sequence, but in any order.

(ANYANS <sequence of strings>)

After space compaction, upper conversion, and word scanning of <response>, if any <string> in <list of strings> matches any word of the student's <response>, then a match occurs. Thus, if the student types an item from the sequence, the answer will be counted as correct even if he or she typed additional words before or after it.

(EXACTANS <sequence of strings>)

this is like ANS except that upper conversion is not performed.

(AFFIRM)

Some "true"-like response must be given (case ignored): Y YES TRUE T.

(NEGATE)

Some "false"-like response must be given (case ignored): N NO FALSE F.

(TRANS <list of strings>)

This opcode relates specifically to the logic course and other mathematically based courses. Each "string" in the <list of strings> is parsed into an internal expression (by the compiler); the student must then type something that, when parsed, is VEQUAL to the parsed expression (VEQUAL is a function in the logic- and set-theory-based programs.)

(MATCHANY)

MATCHANY is an opcode that matches any non-null answer whatsoever. It is useful for the logic of some of the opcodes such as CAS and WAS. For example, in the following Q opcode the author traps two specific wrong answers, responding appropriately to each. For any other wrong answer, the program simply replies by typing "Wrong" on the terminal:

```
(Q      A      (ANS "74")
        WAS    ((ANS "73") (S "One too few.")
                (ANS "75") (S "One too many.")
                (MATCHANY) (T "Wrong.")))
```

2.10 Conditional Opcodes

A number of opcodes check a condition in the driver to decide whether or not to execute the statements contained therein. For example, one wants to do something differently depending on whether or not audio is available.

(AUDIO <actions>)

This executes <actions> if audio is available and selected.

(DPY <actions>)

This executes <actions> if audio is not available or if it was not selected, provided the student is using a display terminal. It is an error to include any audio-class opcode within the scope of a DPY.

(TTY <actions>)

This executes <actions> if the student is using a nondisplay terminal. Both audio- and display-class opcodes are illegal.

(NONAUDIO <actions>)

This executes <actions> if audio is not available or not selected. It is an error to include an audio-class opcode within the scope of a NONAUDIO opcode. And since the intent of the NONAUDIO opcode is to supplant both the DPY and TTY opcodes, display-class opcodes should not be used either. Also, this opcode should not occur in the same exercise in which the DPY or TTY opcode is used.

2.11 Branching Opcodes

(GOTO <lessonnumber> <exercisenumber>)

This goes to <lessonnumber>.<exercisenumber> in the course. If <exercisenumber> is empty, then 1 is assumed. This changes the student's place in the course.

(CALL <lessonnumber> <exercisenumber>)

This interprets an exercise for the student, but does not require any answers from him. The student may not log out from that lesson, but must return to the current "lesson.exercise" before continuation is allowed.

(CHOICELESSON <lessonnumberlist>)

This gives the student a choice of which lesson he wants to do next.

(BRANCHLESSON "text" <lessonnumber>)

The `text' will be typed in the scroll region, then if the student has answered 75 percent of the Q-type exercises in the current lesson correctly when BRANCHLESSON is encountered, he will be given the choice of doing the rest of the exercises in the current lesson or skipping ahead to <lessonnumber>; if he has not answered 75 percent correctly, he must do the remaining exercises.

(RAND <action> <action> ....)

This selects randomly one action to be executed. For example, if we have n <actions>s, then a random number between 1 and n is generated, and the corresponding <action> is performed.

NOTE: RAND is generally used to select a derivation for the student to do. If the student logs out in the middle of such an exercise, then his work is saved. The next time he logs back in, he must get the same exercise. So the randomly selected options picked by the driver are executed in the same order on subsequent logins.

2.12 HELPMODULE Opcode

The HELPMODULE opcode has the syntactic form:

(HELPMODULE <number> "description" <graph info> <actions>)

where <number> is the number of this help module; "description" is a one-line description of this help module, to appear at the top of the display screen when the actions are interpreted; <graph info> is a list of information needed by the help system; and <actions> is a sequence of VOCAL actions for this help module.

There are three types of information which may be placed in the <graph info> list. The first two are mandatory, the last not. They are:

```
(NAME "name of the help module")
(DESC "description of what the help module covers")
(EMPH (<L1> <L2>) (<L3> <L4>) .....(<L2n-1> <L2n>))
```

Both the NAME and DESC opcodes facilitate the student's accessing of a help module, and are used in offering the choice of available topics. The graph info description may be the same as that which is used as a display header, or it may be different. The optional EMPH opcode specifies lessons in which the help module is to be emphasized. Each Li which occurs within the scope of the EMPH opcode is an integer corresponding to a lesson number. These should appear in ascending numeric order. See Section 4 for details of the semantics and a description of the help System.

2.13 Flag-alteration Opcodes

The following opcodes set various entries in the student history file or in the proof checker, and control the run-time operation of the driver. The actual theories and history entries are discussed in course-specific supplements available to lesson authors.

(CONNECT "theoryname")

This selects the context theory (including parser, grammar, axioms, definitions, and theorems) for derivations and student response under the Q and CQ opcodes.

(INTERPCONNECT "theoryname")

This selects the context theory (including parser, grammar, axioms, definitions and theorems) for interpretation-type exercises.

(WANT!DATA "truth value")

This flags the student for collection of data, e.g., connect time, responses, etc. "Truth value" in this and the following opcodes must be either "TRUE" or "FALSE"; that is, "T" and "F" are not sufficient.

(LINE!INFO "truth value")

This selects format for printing proofs during execution of a derive-class opcode ("FALSE" suppresses printing of flagging, dependencies, etc.).

(SHARP!FLAG "truth value")

This selects form for ambiguous names during execution of a derive-class opcode. Value = FALSE requires the student to use ambiguous names of the form #<integer> in existential specifications. Value = TRUE allows the student to use any legitimate variable name.

(ERROR!MSGS "truth value")

Value = TRUE causes error messages concerning quantifier rules to be printed after the student enters a line containing such an error; value = FALSE supresses quantifier error messages and causes the student to be informed at the completion of the proof that it is incorrect due to a quantifier rule error.

The following opcodes allow changes to arbitrary history lists. The first element of a history list is the name of some set, and the remaining elements of the list are the members of that set.

(SET "historyname" ("item" "item" ...))

If there is a history list with the name 'historyname', SET changes its value; otherwise SET creates a new history list of the form (historyname item item ...).

```
(ADD "historyname" ("item" "item" ... ))
```

This adds items to the set of items in the list with name `historyname'.

```
(REMOVE "historyname" ("item" "item" ... ))
```

This removes items from the set of items in the list with name `historyname'. If an "item" mentioned is not on the list, it is ignored. To remove all the items from a list, (SET "historyname" ()) is better than using REMOVE.

2.14 DERIVE-class Opcodes

The following opcodes all specify some sort of call on the derivation machinery. Their interpretation is done by the proof checker and is discussed in course-specific supplements available to lesson authors.

```
(DERIVE "goal" <tag> <action> <tag> <action> ... )
```

"Goal" is the string representation of a formula to be derived. This is a required argument and must immediately follow the opcode. The other arguments to DERIVE consist of pairs of <tags> and <actions>, all of which may be defaulted and may occur in any order.

1. INIT <actions>. These actions will be done when the DERIVE opcode is begun. If the DERIVE opcode is not within the scope of a TEM or TEM2, the <actions> can be replaced by a TEM or TEM2.

2. PREML ("premise" "premise" ...). This is a list of string representations of the premises.

3. HINT <action>. If a hint is requested by CTRL-H, then <action> will be executed.

4. HINTL (<action> <action> ...). Whenever CTRL-H is typed, the next <action> is executed (the last is repeated as often as requested).

5. RESTL <restrictionlist>. This is the list of restrictions on the rules of inference, etc., that the student must or must not use in this derivation.

6. DONE <doneaction>. When the proof has been completed, <doneaction> is executed.

Other opcodes with the same syntax as the above are:

1. INTERP. The student is asked to provide an interpretation (in the connected interpretation theory) of the premises in PREML and the conclusion in "goal".

Then the student is asked to `check' his or her interpretation by deriving the interpreted premises, and the negation of the interpreted conclusion in the connected theory for interpretation.

2. INTERC. This is similar to INTERP, but the student is asked to show that the premises and conclusion are consistent.

3. DI. The student is asked to decide whether to do an interpretation or a derivation. If the former is chosen, the driver interprets this opcode as a DERIVE; if the latter is chosen, the opcode is interpreted as INTERP. The student may restart the exercise and change his or her choice.

4. DIC. This is like DI except the choice is between INTERC and DERIVE (where the student is asked to derive the negation of "goal" if he chooses to do a derivation).

(SYMB "formula" HINT <hintaction> "goal" <tag> <action> ...)

The SYMB opcode first requires the student to symbolize the "formula" (which can also be a list of formulas, ("formula" "formula" ...), in which case he must symbolize all of the formulas). If he requests a hint at this stage by typing CTRL-H, <hintactions> will be executed. The HINTL tag can be coded, with the usual interpretation. If the student does the symbolization correctly, the exercise is ended; if not, he is given the choice of trying again or doing a derivation. If he chooses the latter, the DERIVE opcode is invoked with the "goal" and list of tags and actions coded in the SYMB opcode.

(TA "formula" (<atomic> "value") (<atomic> "value") ...)

`Value' is a truth value (either T or F), and <atomic> is an atomic sentence of the current theory. TA requires the student to determine the truth value of `formula', given the values for the atomic sentences of the formula. This opcode is generally used only within the scope of the TTY opcode, where the display capabilities for conveniently presenting and dynamically altering truth tables are absent.

(CEX "goal" <tag> <actions> <tag> <actions> ...)

Here, the syntax is the same as in DERIVE; it asks student to provide a sentential counterexample by choosing truth values for atomic sentences that will make the truth values of the premises true, the truth value of `goal' false, and then proceeds in each case as in TA, to check if he has succeeded.

(DC "formula")

This is like DI, except the student chooses between doing a derivation (DERIVE) or a sentential counterexample (CEX).

It is an error to have more than one derive-class opcode in the same exercise unless all of them occur within the scope of a MENU opcode.

2.15 MENU Opcode

(MENU <criterion> INIT <actions> DONE <actions> <entree> <entree> ...)

The MENU opcode is designed to provide the lesson author a means of presenting a set of derivations or other derive-class exercises for the student to work on. The student is allowed to select the order of exercises, and need not complete all of them. Work on partially completed derivations is saved. The <criterion> is the number of entrees the student must complete before going on to the next exercise. INIT <actions> are actions to be performed when the student begins the menu (and repeated when he logs in after logging out while in a menu). The <actions> may be replaced by a TEM or TEM2 if desired. DONE <actions> are performed when the criterion is met by the student (i.e., he has completed <criterion> number of the entrees).

Each <entree> has the form:

(ENTREE "description" <action>)

The "description" is a one-line string which will appear in the MENU, and <action> is the exercise for the ENTREE. The <action> is generally a derive-class opcode, although it may also be a TEM or a TEM2 which contains a derive-class opcode. It is an error to include a Q-class opcode in an ENTREE.

The action of the MENU is as follows. First, the INIT actions are performed. Next, a "menu" is displayed which consists of an identifying number for each ENTREE, the string "description" for it, and an indication of whether the ENTREE is completed, partially completed, or not begun. Also the criterion will be displayed, with an indication of how much remains to be done. The student then selects which ENTREE he wants to work on and the <actions> for that ENTREE are performed. When he finishes (or decides not to do any more work on that ENTREE) the MENU is displayed again. When the criterion is met, the DONE actions are performed and the student is then given the choice of selecting more ENTREEs (if any remain) or of going on to the next exercise.

It is an error to have more than one menu in a given exercise.

2.16 Compiler-specific Opcodes

The following opcodes affect only the operation of the compiler and are "swallowed" by it.

(COMMENT "anything")

The above is a comment, at any point it occurs, and is ignored by the VOCAL compiler.

(COMPILECONNECT "theoryname")

This opcode tells the compiler to start using "theoryname" as the context theory (including parser and grammar) for compiling formulas in the scope of derive-class, or TRANS, opcodes.

(TEMCHAR "character")

Resets the character used for the templates to the first character of the string "character". The default character is "%".

3 Browse Mode

There is a feature available in all VOCAL-based courses called Browse Mode, which allows the student to review old exercises or to look ahead in the curriculum. Browse Mode is entered by typing a CTRL-B. The program will reply by typing the following summary of the Browse commands:

```
**** BROWSE MODE ****



Type one of the following:

D n      Describe Lesson `n'
L n      List directory of Lesson `n'
S n      Show Exercise `n' in this Lesson
S l.n    Show Exercise `n' in Lesson `l'
I        Index of Saved Proofs
R l.n.e  Review Proof (lesson.exercise.entree)
E l.n.e  Erase Proof (lesson.exercise.entree)
^Z       Return to your exercise.
```

Here is an explanation of those commands:[9]

D n   Describe Lesson number n. Each lesson has a brief discussion describing the material it covers. This will be the <actions> taken from the LESSON opcode (see p. 4).

L n   List the lesson directory for Lesson n. Recall that each exercise begins with a brief one-line description of itself (the "description" string from the EXERCISE opcode, p. 4). The L command presents all of these at once for the lesson specified.

[9] I, R, and E are specific to the logic course and may or may not be available in other courses.

S n — Show exercise number n in the CURRENT lesson (i.e., the lesson the student was in when he entered Browse Mode). This allows him to look at the exercise just as if he were working it, EXCEPT that anything within the scope of a Q, CQ, or derive-class opcode (including MENU) will not be executed.

S n.l — Show exercise n in lesson l. This is just like the other version of Show, except that it allows the student to look at an exercise in a lesson other than the current one.

I — Index of Saved Proofs. There is a facility available that allows the student to save proofs so that he can look at them again later. To find out what he has saved, the student goes to Browse Mode and types 'I'. This gives him a directory of the available proofs.

R l.n.e — Review Proof (lesson.exercise.entree). Once the student knows the number of the proof he wishes to look at, he uses the R command to actually see the saved proof.

E l.n.e — Erase Proof (lesson.exercise.entree). The student is allowed to store <u>only</u> about 25 proofs for later reference. Once he has reached the limit, he is not allowed to add a new proof unless he deletes an old one first. The Erase Proof Command allows him to do this.

CTRL-Z — Leave Browse Mode. This returns the student to the lesson and exercise he started from. The program saves a copy of his "chalkboard" area and will restore it to what it was just before he entered Browse Mode; the "scratch-pad" area is lost.

Browse Mode makes it easy for the the student to review something earlier in the course, or if he is the kind of person who likes to look ahead in a book, he can do the on-line equivalent. This feature is especially important in courses where the curriculum material is presented at display terminals instead of in textbook format off-line or in classroom lectures. Notice that other than writing the description strings and the <actions> for the LESSON opcode, authors need not make any special effort. It is taken care of automatically by the VOCAL compiler and the course driver.

4 <u>The Help System</u>

A help-system facility is available for use with all VOCAL-based courses. It can be used to create and maintain a help system for the course. This section explains the structure such systems should have, and how to implement them.

4.1 Description

The student invokes the help system by typing 'HELP' to the course driver. When he enters the help system he has two ways of accessing information: by naming the topic on which he needs help or by being questioned by the computer.

The computer queries the student by means of a topic decision graph which is used to present him with a selection of topics to choose among. The particular question or 'interrogation' structure used by the computer is determined by the author. However, two factors can cause additional options to be added to the initial topic choice presented. A relevant help topic will be placed at the head of the topic list if the program is currently in some error mode, or if the author has specified that the topic should be emphasized within the student's current lesson. Thus a syntactic error in an input string might cause a help module on syntax to be emphasized. And in the early lessons of a course, it might be desirable to place emphasis on the help system itself.

The following gives an example of how a choice might typically be presented to the student:

```
Type                      for help on

TA          [TA]          TA duties, hours, etc.
RULES       [R]           Rules of inference
CONCEPTS    [CONC]        Basic concepts of the course
ADMIN       [AD]          Administrative matters
GRIPE       [GRI]         How to send a suggestion or complaint

* <student inputs choice here>
```

The characters in square brackets give the shortest sequence of characters the student can type and have his request properly recognized. But he is not forced to choose one of the options explicitly displayed before him; he can, if he wishes, enter the name of any help module available, provided he knows it. The choices are presented in case he does not know the name of the suitable topic.

Some topics will have associated VOCAL code to be presented as help to the student. Other topics exist only to guide the interrogation process, and have no VOCAL actions written for them. For example, the help system might present the RULES option only to give the student a chance to find out the names of the help modules describing the various rules of inference.

If VOCAL code for a chosen topic is written, the help system first fetches the corresponding HELPMODULE and interprets the VOCAL code in it.[10] The student is then presented with another list of options. If the last chosen topic has further topics associated with it in the

[10] If the student finishes typing in a topic name with a CTRL-Z, only the the associated topic list is typed out. This allows a student who has already seen/heard the VOCAL code to skip over it.

graph, the choice list will consist of these topics, otherwise it will be the initial list. The student may then either exit from the help system (via CTRL-Z), choose another topic, or return to the initial interrogation by entering the null response, i.e., the empty string.

The interrogation procedure should be regarded as a graph rather than as a decision tree. In many cases it is natural to use a topic-subtopic structure. But the author may also want to remind the student of related topics. Thus a module discussing the concept of tautology might have modules on consistency and inconsistency as associated topics. It is also easy and often useful to define loops in the interrogation structure. This will become clearer in examples to follow.

4.2 Defining a Help System

It is necessary to write a series of files that contain the information to be made available via the help system. If the help system is intended to offer information on teaching assistant hours, rules of inference, and tautologies, corresponding help modules must be written in these files: All the help actually given to students is taken from them.

These files are written in VOCAL, where each entry is within the scope of a HELPMODULE opcode (see p. 19). Except for the inclusion of the special tags NAME, DESC, and EMPH, each entry looks just like an ordinary exercise, with the EXERCISE opcode replaced by the HELPMODULE opcode. The help system files cannot contain the opcodes LESSON or EXERCISE, but they can contain questions and derivations designed to help a student understand the topic. The student's performance on these questions will not affect the 'score' which is recorded for control of BRANCHLESSON.

A special file needs to be written to define the interrogation structure used by the computer. This file will contain a graph which is usually easily written and which is easy to alter if experience suggests that another choice sequence is more appropriate. The help system compiler is then run to put all this information in a form acceptable to the course driver.

4.2.1 The Help Module Files

The help modules may be included in one or more files. The source files may be named as desired, though they should all have a .VOC extension for ease in operating the VOCAL compiler. Each such file contains a sequence of HELPMODULE opcodes. They furnish information used in the interrogation, and provide the VOCAL code to be interpreted if help on a particular topic is requested by the student. The helpmodule files must be processed by the VOCAL compiler, and the resulting .VAL files submitted to the help system compiler.

If different help systems are being defined for several courses, they may share some of the same helpmodules. Help on the use of Browse Mode, for example, would probably be the same for all courses. For this reason, the .VAL files for helpmodules, unlike those for lessons, do not have to be on the curriculum directory. The HELP.PRC file which is created for each course by the help system compiler informs the driver of the directory and file name on which the helpmodule for each topic defined in the graph may be found. A help system does not have to use all the helpmodules on any file which it accesses. Different courses may 'pick and choose' among the various modules on a given file. If, however, the same topic name is to be used with different help information in different courses, separate helpmodule files must be created for those topics whose definitions vary from course to course. This might occur, for example, in writing helpmodules on inference rules which have the same names for each course, but use different derivation examples in the theory appropriate to the course.

The syntax and semantics of the HELPMODULE command (p. 19) should be clear, with the possible exception of the optional EMPH opcode. This is used to indicate the lessons in which the help module needs to be emphasized in the sense of being placed at the head of the topics in the initial interrogation of the student.

To illustrate this, suppose that it is desired to define a help module about some derivation command, e.g., Tautology. Suppose in addition that this rule of inference is introduced in Lesson 76 and that exercises in Lesson 105 are particularly suitable for its application. One might then write the following help module:

```
[HELPMODULE 3 "TAUTOLOGY: Using the Tautology Rule"
  ((NAME "TAUTOLOGY")
   (DESC "Using the Tautology Rule")
   (EMPH (76 78) (105 105)))
  [AUDIO
   (S "Just type 'T' to use the Tautology Rule")]]
```

Here the help available on the Tautology Rule would be emphasized in Lessons 76 through 78, and 105.

If a helpmodule is to be accessed by more than one course, the EMPH list may contain lesson numbers from each of these courses. For example, if there are three courses whose lesson numbers range respectively in the 100's, 200's, and 300's, then a topic which is to be emphasized in the beginning lessons of each course might have the EMPH list: (EMPH (101 102) (201 202) (301 302)).[11]

If EMPH strings are declared for any help modules, the help system compiler will produce a listing of the emphasized topics by lesson.

[11] If the lesson numbers for two courses are the same, and a helpmodule is to be emphasized in different lessons in these courses, it will be necessary to make separate versions of the helpmodule with different EMPH lists.

Consistent with the rest of the VOCAL language, the help modules in any file must be numbered in increasing order, from 1 onward. Furthermore, the length of module NAMES should be 11 characters or less. Longer strings will disorder the display output, although not fatally. Similarly, the length of DESC strings should not exceed 46 characters. The length of the help module names is not critical apart from aesthetic considerations, since optimal topic name recognition is done automatically.

The help system compiler does extensive error checking, and gives suitable warnings. For example, the author will be warned if he doubly-defines a help module topic or fails to define a HELPMODULE for a topic declared in the graph. Note that helpmodules must still be written for decision nodes which have no associated VOCAL actions, since the helpmodule contains the description information (needed for the interrogation display), and possibly emphasis data.

4.2.2 The Graph File

Only one file is needed to define the interrogation structure for any course. This file should have the extension .GRAPH, and may be on any directory. When the help system compiler is run, it looks on the connected directory for the graph file unless another directory is specified. It also uses the name HELP.GRAPH as a default. The file should contain a single S-expression, which is essentially a list of topic names and their subtopics. The S-expression is recursively defined as follows:

```
graph S-expr    ::=  (<topic-list>)
topic-list      ::=  <topic> <topic list> | <null string>
topic           ::=  <topic name> | <topic name> <graph S-expr>
```

The following is an example of what a help system list for an elementary logic course might look like. It incorporates all the above illustrations.

```
[TA
 RULES
          (BOOLE
           TAUTOLOGY
           HYPOTHESIS (ANTECEDENT)
           AA
           CP (CONDITIONAL))
 CONCEPTS
          (ANTECEDENT
           CONSEQUENT
           TAUTOLOGIES
                (CONSISTENCY
                 INCONSISTENCY)
           ALGORITHM
           CONDITIONAL (CP)
           CONSISTENCY
           INCONSISTENCY)
ADMIN
        (TA
         NEWS
         GRIPE
         EXAMS
         GRADES)
GRIPE]
```

The central idea is that options which are to be displayed at the same interrogation are at the top level of the same list, and the topics associated with a topic are contained in a list immediately following the topic. Thus the first level of interrogation in this particular help system would be between TA, RULES, CONCEPTS, ADMIN, and GRIPE, just as previously illustrated. Further interrogation is defined by the sublists. So if GRIPE is chosen, the helpmodule, as defined in the helpmodule file, is interpreted, telling the student how to send suggestions and complaints to the course authors; if RULES is chosen, the VOCAL code (if any) in the helpmodule files is interpreted, and the further subtopic list of topics is presented.

As previously remarked, the interrogation need not be a tree. In the above example there are two paths to the same node for both the CONDITIONAL and the GRIPE help modules. In addition, there is a cycle: If information on the CP rule is given, the student will be reminded of the help available on the concept of a conditional; if this latter help module is interpreted, the student will then be reminded of the help on the CP rule.

4.2.3 The Help System Compiler

When the files containing the interrogation graph and the help modules have been written, the next step is to process them. One begins by processing the HELPMODULE files using the VOCAL compiler as described in the next section. VOCAL will create processed versions of the help module files. These files will have a .VAL extension, just as for lesson files.

The help system is created by running the help system compiler, called HELPMK for 'HELP system MaKer'. This program processes the graph and help module files, producing a corresponding run-time file, HELP.PRC, for use by the curriculum driver. It is self-explanatory in operation and includes extensive error-checking and analysis.

When one calls HELPMK, it first presents the option of creating a new system from scratch, or modifying an old one. If the Create option is chosen, the graph file is first processed, and then helpmodule files are processed until all the topics in the graph have been defined. Under this option, when the compiler encounters a helpmodule for a topic which has already been defined by an earlier helpmodule, the second and subsequent definitions are ignored, and a warning is printed. The help system is ready when the HELP.PRC file has been moved to the curriculum directory.

If a fatal error occurs after the graph file has been processed (for example, some module has incomplete graph info), then the structure at that point is dumped onto a file called HELP.INCOMPLETE and the compilation is aborted. The error may then be corrected, and HELPMK run with the Modify option using HELP.INCOMPLETE as the 'old' system.

If topic emphases have been declared, a further file, HELP.LESSON-EMPHASIS, is also created, giving a listing by lesson number of the emphasized topics. This will assist the author in keeping track of which topics are emphasized in which lessons.

4.2.4 Modifying an Existing Help System

If one wishes only to change the VOCAL actions in a helpmodule which is already defined in a working help system, it is sufficient to edit the source file and generate a new version of the .VAL file (the file name and directory must not be changed). The HELP.PRC need not be regenerated; it will always call up the newest version of the helpmodule file.

If however, one wants to change the lesson emphasis or other graph info in a module, if the .VAL files are renamed or moved, if new modules are to be added, or if the graph structure is changed in any way, the changes must first be made in the graph and/or helpmodule files, and then HELPMK must be run to create a new version of HELP.PRC.

Under the Modify option, only the new files or those which have been changed need be processed. Unlike the procedure in Create mode, second occurrences of a helpmodule for a topic will not be ignored but rather will cause the module to be redefined. The system will retain all information from the old system which is not superceded by the new files.

Note that to delete a topic from the system, it is sufficient to remove it from the graph S-expression, and run HELPMK with the Modify option over the new graph file, without reprocessing the helpmodule file. The helpmodule will simply exist on a .VAL file but never be accessed. To add a topic to the system, however, both the graph file and a helpmodule file must be edited and reprocessed.

5 Operation of the VOCAL Compiler and Interpreter

Once a lesson has been written, it must be tested and compiled. This is done using the VOCAL compiler, a program that assists authors in lesson testing and debugging and that produces various files needed for the audio system and course driver. This section describes the operation and features of VOCAL.

5.1 Initializing VOCAL

The lesson source files are usually given names of the form "<lesson>.VOC". The exact form of <lesson> depends on the course.[12]

When VOCAL is started, it prints a herald giving the program name and a "version number". The latter consists of the date and time that this version of VOCAL was compiled. Then, before doing anything else, VOCAL attempts to assign the audio channel associated with the controlling terminal. If there is none or if VOCAL is unable to assign it, it will say, "No audio channel assigned." If the MISS machine is down, it will also say, "Audio system not available." In any case, if the channel is assigned, VOCAL begins in AUDIO mode. If the audio channel is not assigned, VOCAL will begin in DPY or TTY mode according to the type of the controlling terminal. In these modes authors are subject to the VOCAL conditional opcodes AUDIO, NONAUDIO, DPY, and TTY, described in Section 2.10. For example, in AUDIO mode VOCAL will not execute any actions surrounded by NONAUDIO, DPY, or TTY opcodes—and similarly for the other modes. This can be changed by use of the A, Y, or TTY commands described below.

If VOCAL succeeds in assigning the audio channel for the controlling terminal, it will ask, "Do you want Long Sounds?", followed by an "*", which is VOCAL's prompt character at this level. If the lesson has been recorded, the answer will usually be "Y", otherwise "N" for "no" or just a carriage-return which defaults to a no answer. If the answer was no, VOCAL next asks, "Do you want PROSODY?" A "Y" answer to this question means that any audio messages will be spoken using the prosody system. A carriage-return again defaults to a no answer. If no audio channel was assigned, these two questions will be skipped.

5.2 Top-level VOCAL Commands

After initializing, VOCAL will print "Type T, I, G, or ? *". This is the top command level for VOCAL, and there are numerous options available:

? Print a list of the available options.

T Enter Test mode. This mode allows the author to test a lesson using code which has not yet been compiled. Test mode is described in Section 5.3.

I Enter Interpret mode. This mode is much like test mode, *except* that VOCAL will read files that have already been compiled. See Section 5.4 for details.

G Enter Generate files mode. This is the mode which allows

[12] For Stanford's logic course, the lessons have names of the form Lnnn.VOC, where nnn is a unique three-digit number assigned to the particular lesson. Other courses may have different conventions.

an author to compile a lesson. See Section 5.5 for details.

S Adjust the Speech rate of the audio system. In the current implementation, the default speech rate is 0.9. This may be adjusted to anywhere between 0.25 and 1.2. A small number increases the speech rate and a larger number slows it down. This has an effect only when testing or interpreting a lesson in audio mode with an audio channel assigned.

A Select Audio mode. If an audio channel has been assigned, VOCAL will repeat the two questions concerning "Long Sounds" and "Prosody". This provides a faster way of switching between the three modes than an exit to the EXEC to obtain a fresh core image. If no audio channel is assigned, VOCAL will be set in a mode where if a lesson is tested or interpreted in audio mode, the text of whatever would have been spoken will print out in the SCROLL region of the terminal prefaced by the word "SPEAK:".

Y Select displaY mode. If a lesson is tested or interpreted, the code within the scope of a DPY opcode will be executed.

TTY Select TeleTYpe mode. If a lesson is tested or interpreted, the code within the scope of a TTY opcode will be executed.

ERR Interrupts the program with a pseudo user error in order to enter DDT or other debugging program. This feature is for the convenience of the programmers and is not normally needed by the lesson authors.

X Make a graceful eXit from VOCAL. Preferable to a ^C exit, since all open files will be properly reset and closed.

5.3 T--test VOCAL Files Mode

When an author enters Test mode, VOCAL will begin by asking for a file name. If the author wishes to be able to use the TVEDIT text editor on the file while inside VOCAL, the file must be on the connected directory. No filename recognition is done, but if the extension is the standard ".VOC" only the name field needs to be typed.

Once a file has been sucessfully opened, VOCAL begins prompting with "==>" instead of "*". In test mode, VOCAL files can be compiled and tested one "object" (i.e., S-expression) at a time. No output files are generated. The audio channel (if assigned) is open in "spell" mode (unless reset at command level using the "A" option), so that lessons can be tested without their Long Sounds or prosody.

The commands available in Test mode are:

? Print a list of the available options.

J Jump over page marks. The author is asked how many to jump. No compiling is done and progress through the file is faster than with "G". Random access is not available at this stage, which limits the use of "J" to going forward. Jumping "0" pagemarks resets the file to page 1.

E Edit and Look Ahead. As an aid to debugging, a simplified scanner is provided to "look ahead" on the current page. The scanner prints out the text (suppressing the content of strings) and checks for delimiter mismatches and misspelled opcodes. TVEDIT, the IMSSS text editor, is available in this mode to alter the VOCAL text. Upon return to Test mode, the file is reset to the beginning of the current page (thus a "G" must be executed to get and compile an object from it). The current page may be changed by using the "J" command. Upon entering the scanner the author is prompted with "TV or Look". If the author then provides a "list of opcodes", the current page will be scanned for each opcode in the list, in order. The delimiter scan will commence with the last opcode in the list. The default is the first opcode on the current page, and typing TV (carriage-return) takes the author directly to the editor.

G Get next object (S-expression) from file and compile it.

I Interpret current object. NOTE: The interpreter will not interpret any derive-class opcodes, nor any branching or flag-altering opcodes.

D Display Current Coding. The current object (S-expression) is typed out for inspection.

C Compile current object (S-expression). The current object is processed so that it can be interpreted using the I command.

O Display Current Output Coding. The compiled version of the current object is typed out. Note that this is not everything that goes into the final output file (.VAL file) as a result of the "G" mode below. See the discussion of the file format in Appendix A.

S Adjust the speech rate. Same as the top-level S command.

A Select Audio mode. Same as the top-level A command.

T Select Teletype mode. Same as the top-level TTY command.

Y Select Display mode. Same as the top-level D command.

P Select Prosody mode. Invokes the prosody system.

Q Select nonprosody mode. Resets the audio system from prosody mode to spell mode.

L Select audio symbol letter. Useful when looking at a file in which recorded audio is present. By selecting the letter(s) used in the file, an author can hear all recorded messages while obtaining unrecorded messages in Spell mode.

LANGUAGE Select an audio language. this allows an author to change languages without restarting VOCAL.

ERR Interrupts the program with a pseudo user error. Same as the top level ERR command.

X EXit Test mode. Returns control to the top level. Note that to completely exit VOCAL from inside Test mode, it is necessary to type two X's.

5.4 I--Interpret Previously Compiled VOCAL Files Mode

Once a lesson file has been compiled, it can be interpreted using VOCAL by entering Interpret mode. Just as in Test mode, VOCAL begins by asking for a file name. However, the default extension is VAL instead of VOC. All VOCAL opcodes will work correctly in this mode, except that, as in test mode, derive-class, branching, and flag-alteration opcodes are not interpreted at all. VOCAL will usually detect major syntax errors within the scope of these opcodes and warn the author. But there are many kinds of errors that it cannot detect. So it is very important that a compiled file is tested using the appropriate curriculum driver before the lesson is made available to students.

J Jump to an Exercise. VOCAL then asks for the number. Once the lesson has been compiled, VOCAL can randomly access the exercises; thus there is no restriction about going backward or forward in the lesson.

I Interpret Current Exercise.

G Go to next Exercise. Goes to the next higher number exercise. If there is no next higher numbered exercise, VOCAL will print a message to that effect.

D Display Current Compiled Code. Prints out the compiled S-expression for inspection. Note that this is only a part of the compiled VOCAL file. See the description of the file format in Appendix A.

S Adjust the speech rate. Same as the top-level S command.

A Select Audio mode. Same as the top-level A command.

T Select Teletype mode. Same as the top-level TTY command.

Y Select Display mode. Same as the top-level D command.

ERR Interrupts the program with a pseudo user error, in order to enter DDT or other debugging program.

X EXit Interpret mode. Returns control to the top level. Note that to completely exit VOCAL from inside Interpret mode it is necessary to type two X's.

5.5 G--Generate Compiled Files and Audio Scripts Mode

When an author enters Generate Files mode, VOCAL begins by asking for a filename. Like Test mode, the default extension is VOC. Next, VOCAL will ask which version to compile: audio (A), display (Y), or teletype (TTY). If audio is selected, the author has a choice of prosody or 'Long Sounds'. If Long Sounds are selected, VOCAL will ask for a letter to precede the lesson number in the sound names in the script file. This letter is used in making up the internal "name" of each Long Sound.[13] VOCAL will also ask whether it should retain the audio text in the VAL file. This allows the file to be interpreted in Spell Mode. Normally, the audio text should not be retained, but this is sometimes useful when the audio messages are going to be recorded and an author wishes to use the curriculum driver to test the lesson before the recordings are available.

When a lesson file is compiled, the following files are produced:

.VAL file. This is the file interpreted by the curriculum driver or by VOCAL in "I" mode. The internal file format is shown in Appendix A.

.SCP file. This is the audio script which is used to record the "long sounds" for the lesson. It is simply a list of pairs of sound names (also called "audio symbols") and the text to be spoken.

.BRA file. If PROSODY is selected, this file contains the spoken material in "bracketed" form. The bracketing represents the prosody module's idea of the syntactic structure of the material to be spoken.

.LST file. When compiled in Long Sounds mode, this file has the long sound names inserted in the "SN" opcodes. It may be useful if part, but not all, of the lesson is to be revised. In that case, the SN opcodes can be changed back to S opcodes in the revised portion (deleting the sound names) and the lesson can be recompiled. No audio script will be generated for the unchanged portions of the lesson. When compiled in prosody mode, this file contains

[13] Authors should see the supplement for their course to find out which letters are currently being used.

the bracketed strings in place of the original audio text. With careful rebracketing, this file can be used to recompile for prosody.

Appendix A

File Format for Compiled VOCAL Files

The file format for TEMPLATES is:

```
WORD 0: TOTAL   AREAS DEFINED (A2LAST)
          AREAS IN TEM1 REGION (A1LAST)
        TOTAL   OF STRINGS (I2LAST)
          LINES IN TEM1 REGION (I1LAST)
        LENGTH OF STRINGS FOR ARRYIN/ARRYOUT (TEMSLENGTH)

        COMPRESSED AREA DEFINITION TABLE (AREADEFS)
        RELOCATABLE BYTEPOINTERS FOR STRINGS (TLINES)
        COMPACTED STRINGS BUFFER (TEMSTRINGS)

        <S!EXPRESSION FOR AREA ASSOCIATION> (AREALIST)
```

The file format for EXERCISES is:

```
WORD 0: <TEMPLATE  1>
          ........
        <TEMPLATE  N>
        <S!EXPRESSION FOR GRAPH INFO>
        <S!EXPRESSION FOR EXERCISE>

INDEX:  EXERCISE
          OF TEMPLATES
        WORD   OF TEMPLATE  1
            ...............
        WORD   OF TEMPLATE  N
        CHARACTER   OF EXERCISE S!EXPRESSION
        CHARACTER   OF GRAPH INFO S!EXPRESSION
```

The file format for LESSONS is:

```
WORD 0: WORD   OF INDEX
        <LESSON DESCRIPTOR>       — "EXERCISE 0"
        <EXERCISE  1>
          ........
        <EXERCISE  N>
        <LESSON DIRECTORY>        — TEMPLATE FOR BROWSER

INDEX:  LESSON
          OF EXERCISES
        WORD   OF LESSON DESCRIPTOR
        WORD   OF EXERCISE  1 INDEX
          ................
        WORD   OF EXERCISE  N INDEX
        WORD   OF LESSON DIRECTORY
```

Appendix B

Implementations of the S Opcode

Currently three implementations of the S opcode are being considered: "Long Sounds", "Prosody", and "Spell Mode". "Long Sounds" means that the "messages" are recorded in segments of no more than 4.5 seconds duration. If an individual "message" is expected to be longer than 4.5 seconds, the lesson author must divide it into two or more messages. "Prosody" will parse the "messages" into a bracketed string ("Arvin-expression"), then generate a list of prosody specifications ("SIL-expression") which will be passed to the audio system for interpretation by the MISS machine. "Spell Mode" is the lesson-author test mode in which the "messages" are simply spoken word by word, and words not available in the audio lexicon are spelled out.

As a further aid to timing, a period encountered within a "message" will be followed by 500 ms of silence, any colon or semicolon will be followed by 250 ms of silence, and any comma will be followed by 125 ms of silence.[14]

The compiler produces a variety of internal forms for the S opcode, which may be retained by the author if he uses the ".LST" output file as his source file. This may be useful in the case of "Prosody" as long as parsing and prosing are slow processes. The syntax of these internal forms is indicated below, but is actually somewhat more complex and subject to continuing development.

(SN "name" "text of message")

The SN opcode is one of the opcodes into which the S opcode may compile. "Name" is the name of a Long Sound (an entry in the audio lexicon). "Text of message" is the original text of the message, and is dropped in the compiled form (unless explicitly retained). This text is always retained in the listing form (which may become the source file).

(SB "bracketed text")

"Bracketed text" is a prosody expression (an "Arvin-expression") that is to be prosodied and spoken.

(SL "literal SPEAK text" "original text")

"Literal SPEAK text" is the "SIL-expression" that should be passed directly to the SPEAK audio library procedure. "Original text" is the original text that produced this and is dropped in the compiled form but retained in the source listing.

[14] The actual duration of silences is subject to the implementation and may change.

References

Davis, M. & Pettit, T., Using VOCAL: A Guide for Authors. (Tech. Rep.). Stanford Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, forthcoming.

Sanders, W., Benbassat, G., & Smith, R., Speech synthesis for computer assisted instruction: the MISS system and its applications (Tech. Rep. #268). Stanford Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1976.

Tesler, L. PUB: The document compiler (Operating Note 70). Stanford Calif.: Stanford University, Stanford Artificial Intelligence Project, 1972.

Index