Prentice-Hall: Englewood Cliffs, NJ, 1971.
(25) Willett, P. "The Calculation of Intermolecular Similarity Coefficients Using an Inverted File Algorithm". *Anal. Chim. Acta* **1982**, *138*, 339-342.
(26) Crouch, D. "A File Organisation and Maintenance Procedure for Dynamic Document Collections". *Inf. Process. Manag.* **1975**, *11*, 11-21.
(27) Willett, P. "Document Clustering Using an Inverted File Approach". *J. Inf. Sci.* **1980**, *2*, 223-231.

Development of a Computer Language and Compiler for Expressing the Rules of Infrared Spectral Interpretation

GRAHAM M. SMITH* and HUGH B. WOODRUFF
Merck Sharp & Dohme Research Laboratories, Rahway, New Jersey 07065

Received July 18, 1983

A specialized computer language, compiler, and interpreter (CONCISE: Computer Oriented Notation Concerning Infrared Spectral Evaluation) were developed to be used to express rules for IR spectral interpretation in the PAIRS[1a] system (Program for the Analysis of Infrared Spectra). The factors considered in designing this language will be discussed as will its final structure. The nature and operation of the associated compiler and interpreter will also be described.

INTRODUCTION

An experimental computer program (PAIRS) has been developed to provide assistance in the interpretation of infrared spectra. In a previous paper[1] the overall design and application of this program was described. The intent here is to provide technical detail on the nature of the specialized language (CONCISE) developed for describing the empirical rules of infrared spectroscopy and on the compiler and interpreter for this language.

The objective of this project was threefold: first, to develop a method to extract, efficiently and consistently, structural information from infrared spectra and to present this information to the user with a calculated level of confidence; second, to provide a means for organizing and storing the empirical rules of infrared spectroscopy; third, to provide a program which is easy to use and rules which are easy to modify. The methods and principles which have been developed will be reported.

An earlier program, CASE,[2] indicated that computer-assisted infrared interpretation was a viable approach; however, it had a major drawback in that the rules were encoded in the logic of the program, which made them difficult to identify or change. A similar problem had been faced by the developers of such programs as LHASA,[3] SECS[4] (organic synthetic planning), and MYCIN[5] (medical diagnosis), and the approach used in these systems was to separate out the rules from the mechanism for interpretation and supply them as data to the programs. In addition, the data in each case were represented in the form of statements of a specialized language which allowed higher level concepts to be directly expressed. This allowed easy understanding of the rules and increased the efficiency of rule growth and correction. It was decided to use this approach in PAIRS.

The IR interpreter program (PAIRS) could have been designed to read the text form of the language directly during the interpretation process; however, this would have caused additional overhead and slowed the response time of what is an interactive program. It was decided to have an intermediate step which would convert the text form of the language into a numerically encoded form which could be read and used quickly. In addition, the text form would only have to be reconverted in the event of modification of the rules. The resulting "conversion" program is in fact a compiler and was developed by using principles of compiler design.[6] The development of this project followed five major steps: (1) Development of a language, starting with the types of information it must be able to express and progressing through exact definitions of statements in a formal grammer[7] and ultimately a parse table. (2) Development of a compiler using the parse table and a definition of the numerical codes. This was followed by design of methods for syntactic analysis and text handling. (3) Development of the main IR interpreter program. This consists of three main parts: the interaction handler, to allow the user to supply information and obtain results; the internal data storage format; the analysis section which would take the spectrum, read the rules, and produce the analysis. (4) Production of the rules as provided by various sources of IR information by beginning with an array of facts about an individual functional group and conversion of these facts into a logical procedure for encoding in the language. (5) Testing of each component in stages and as a unit to ensure that all flaws had been found.

LANGUAGE DESIGN

Language development began as discussions of the process of IR analysis with an expert in this field in order to determine the types of information needed. In the very beginning it was clear that there would have to be a facility to ask questions about peak location, intensity, and width. However, there were many features of human IR interpretation which were not clearly defined to us. It was decided to follow the process of interpretation by an experienced spectroscopist who would verbalize his thinking and stop to answer questions. This was not done to gather exact information for the rules themselves but to understand the process of interpretation. From this analysis several principles were derived about the interpretation process. We also identified a difference between the approach used by the spectroscopist and that planned for use in the PAIRS program. The difference is that on viewing an IR spectrum, various peaks immediately suggest groups to the spectroscopist for which he or she then gathers support among the remainder of the peaks from their position, intensity, and width. During the process other groups may be suggested, and the process continues to a self-consistent conclusion. This appears to be a very complex process. In the case of the computer, however, we implemented a sequential process where each set of rules for each encoded group is applied in turn to the spectrum, giving an analysis for each group. This difference in approach should ultimately not be significant. Several of the principles which emerged from this process will now be discussed. An important point is that questions are not asked by the spectroscopist in a simple series but can be interdependent and deeply nested. The conclusions drawn about one group may depend on conclusions about another group as well as on information from the spectrum. Knowledge of the empirical formula which can be used directly or in connection with spectral evidence can be used to eliminate some groups.

It also appears that the search for evidence for certain groups is dependent on the evidence for other groups, and so the search should be ordered. For example, the presence of a C–O stretching peak may indicate a number of functional groups, alcohol, acid, ester, or ether. Ether is the most difficult to identify and is deemed to be unlikely unless the other groups can be ruled out with good confidence.

The solvent can affect the conclusions about certain types of functional groups where solvent peaks would overlap peaks for the group. It is known that the environment of a group will affect its behavior in the IR; hence, there is a need to provide for tolerances or ranges to be associated with peak positions and intensities. From these observations, it appeared clear that four types of questions were needed, relating to (1) peaks (specification of an acceptable frequency range for peaks: number of peaks in range, intensity range for peaks, width range for peaks), (2) other group probabilities derived earlier in the process, (3) empirical formula (types and numbers of atoms present in the original empirical formula and in the reduced empirical formula), and (4) solvent type.

One point in particular about the peak question was whether accommodation should be made for "shoulders" on peaks. It was decided that a shoulder is caused by partially overlapping peaks and, if detected, should be treated as two ordinary peaks. It seemed most natural that these questions be reformulated as statements which were either "TRUE" or "FALSE".

Also needed were actions which could be used to adjust the developing probabilities and a copy of the empirical formula. Four types of action were decided on: (1) To establish or modify (using arithmetic) a level of certainty or "probability" for each functional group type being considered. The "probability" was defined in the range 0.0 (to indicate certainty that a particular group does not exist) to 1.0 (to indicate certainty that a particular group does exist). In addition, a facility for expressing the possibility that no determination for a group can be made is provided in the term "UNKNOWN". This condition may arise when a group's peaks are masked by a particular solvent. (2) To allow a working copy of the empirical formula to be reduced by the formula for a functional group if the probability for that group rose above a predefined level. (3) To terminate further consideration of the rules for a particular group. (4) To continue operation of the decision tree by taking no action. This is required in cases when there is no desired action needed in a "THEN" clause (the "THEN" must follow the "IF" statement in the first version of the language).

Finally, a control mechanism was needed to allow statements to be nested and therefore used conditionally. There were two chief methods which could have been used to provide the required control. One would have been the "GO TO" type statements with statement labels. An additional facility this would have provided is the ability to perform iteration. The main drawback to the "GO TO" facility is the possibility for the accidental formation of "infinite" loops by even experienced programmers. Also, the excessive use of "GO TO" statements can lead to obscure logic in a program and make understanding and modifying the rules very difficult. It was decided not to use the "GO TO" because of this possibility and because no need for iteration was foreseen. The method chosen to provide control is the "IF–THEN–ELSE" structure. In this method, if a given statement is true, the "THEN" clause is used, and

Chart I

```
IF STATEMENT-1
THEN BEGIN
ACTION-1

IF STATEMENT-2
THEN BEGIN
ACTION-2
ACTION-3
DONE

ACTION-4
DONE

ELSE BEGIN
ACTION-5
ACTION-6
DONE
```

Chart II

```
STATEMENTS:

     IF 1 INTENSITY 4 TO 10 SHARP TO AVERAGE PEAK IS IN RANGE 2750 TO

     2530. . .

     IF ALDEHYDE IS LESS THAN 0.5. . .

     IF 2 OXYGEN REMAIN. . .

     IF SPECTRUM RUN IN OIL. . .

COMMANDS:

     SET ALDEHYDE TO 0.25

     ACID UNKNOWN

     ADD 0.10 TO ALCOHOL

     MULTIPLY KETONE BY 2.0

     SUBTRACT C 1 O 2 FROM FORMULA

     CONTINUE
```

the "ELSE" clause is ignored. If the statement is false, the "ELSE" clause is used, and the "THEN" clause is ignored. In its simplest form, only one statement may be included in each "THEN" or "ELSE" clause. This restriction is overcome by the use of terms which serve as parentheses to enclose blocks of statements which belong to a "THEN" or "ELSE" clause. Here the terms "BEGIN" and "DONE" were chosen. The example in Chart I shows how these statements appear.

At this point, the informal formulation of the statements from the analysis of IR requirements was converted into the definition of a formal language. The first step was the expression of the questions as clear "true/false" "IF" statements and the actions as simple commands (see Chart II).

Some of these statements have several forms to allow for options. For example, the "PEAKS" statement has several forms which allow a specific number of peaks to be required, "AT LEAST" a certain number of peaks to be required or "ANY" number to be allowed. Also, a single width may be specified (e.g., "SHARP") in place of the range "SHARP TO AVERAGE". The details of these options may be obtained from the grammar or by review of the rules written in the language.[8]

Several other technical decisions were made at this point to put the language into final form. The first was based on the realization that there could ultimately be a large number of rules and that to make the information manageable a facility should be provided for organizing the rules for each functional group into individual "routines". This was accomplished by creating four new statement types: HEADER (GROUP-NAME, EMPIRICAL FORMULA, INDEX), END, FINISH, and COMPLETE.

The header defines, by name, a major new group type and gives its empirical formula. The first operation performed when starting this "TREE" (derived from binary decision *TREE*, which is the form of this language) is to compare the formula from the header with that given with the spectrum

```
$
 ALDEHYDE    C 1 O 1 H 1 UNSAT 1     2ØØØ
$
 IF ANY INTENSITY 7 TO 1Ø SHARP TO BROAD
 PEAKS ARE IN RANGE 1765 TO 166Ø
$
     THEN BEGIN
     IF ANY INTENSITY 1 TO 1Ø
     SHARP TO BROAD PEAKS ARE IN
     RANGE 275Ø TO 266Ø
$
         THEN BEGIN
         SET ALDEHYDE TO Ø.25
         IF ACID GREATER THAN Ø.9Ø
$
             THEN BEGIN
             SUBT Ø.15 FROM ALDEHYDE
             DONE
$
             ELSE BEGIN
             IF ANY N EXIST
$
                 THEN BEGIN
                 IF ANY INTENSITY 4 TO 1Ø
                 BROAD PEAKS ARE IN
                 RANGE 275Ø TO 23ØØ
$
                     THEN BEGIN
                     SUBT Ø.15 FROM ALDEHYDE
                     DONE
                 DONE
             DONE
         DONE
$
         ELSE BEGIN
         IF ANY INTENSITY 1 TO 1Ø
         SHARP TO BROAD PEAKS ARE IN
         RANGE 2679 TO 266Ø
$
             THEN BEGIN
             SET ALDEHYDE TO Ø.Ø5
             DONE
         DONE
     IF ANY INTENSITY 1 TO 1Ø
     SHARP PEAKS ARE IN
     RANGE 275Ø TO 266Ø
$
         THEN BEGIN
         ADD Ø.25 TO ALDEHYDE
         DONE
$
         ELSE BEGIN
         IF ANY INTENSITY 1 TO 5
         AVERAGE PEAKS ARE IN
         RANGE 275Ø TO 266Ø
$
             THEN BEGIN
             ADD Ø.25 TO ALDEHYDE
             DONE
         DONE
     IF SPECTRUM RUN IN OIL
$
         THEN BEGIN
         CONTINUE
         DONE
$
         ELSE BEGIN
         IF ANY INTENSITY 1 TO 1Ø
         SHARP TO BROAD PEAKS ARE IN
         RANGE 285Ø TO 28ØØ
$
             THEN BEGIN
             ADD Ø.25 TO ALDEHYDE
             DONE
         DONE
     IF ANY INTENSITY 6 TO 1Ø
     SHARP TO BROAD PEAKS ARE IN
     RANGE 1739 TO 1681
$
         THEN BEGIN
         ADD Ø.25 TO ALDEHYDE
         DONE
     IF ALDEHYDE GREATER THAN Ø.ØØ
$
         THEN BEGIN
         IF ANY INTENSITY 7 TO 1Ø
         SHARP TO BROAD PEAKS ARE IN
         RANGE 1739 TO 1715
$
             THEN BEGIN
             SET ALDEHYDE-SATURATED TO ALDEHYDE
             MULT ALDEHYDE-SATURATED BY Ø.9Ø
             DONE
$
             ELSE BEGIN
             SET ALDEHYDE-SATURATED TO ALDEHYDE
             MULT ALDEHYDE-SATURATED BY Ø.1Ø
             DONE
         IF AT LEAST 2 UNSAT EXIST
$
             THEN BEGIN
             IF ANY INTENSITY 7 TO 1Ø
             SHARP TO BROAD PEAKS ARE IN
             RANGE 1714 TO 166Ø
$
                 THEN BEGIN
                 SET ALDEHYDE-UNSATURATED TO ALDEHYDE
                 MULT ALDEHYDE-UNSATURATED BY Ø.9Ø
                 DONE
$
                 ELSE BEGIN
                 SET ALDEHYDE-UNSATURATED TO ALDEHYDE
                 MULT ALDEHYDE-UNSATURATED BY Ø.1Ø
                 DONE
             DONE
         DONE
     DONE
$
 FINISH
$
 END
$
 COMPLETE
```

Figure 1. Example of the aldehyde interpretation rules encoded in CONCISE.

(if available) to determine if there are enough of each required atom for this group to be possible. If not the entire TREE is skipped. The "END" statement signifies the end of a tree. The "FINISH" statement terminates consideration of all rules up the next "END", similar to a FORTRAN "RETURN" statement. Last, the "COMPLETE" statement signifies the end of the information. It is used to allow experimental TREES to be placed at the end of a file and turned on or off by simple removal or insertion of an extra "COMPLETE". An example is as follows:

```
ACID C 1 O 2 H 1
"IF" statements and commands relating to the acid group
  including various subtypes (e.g., alpha-beta unsaturated acids)
END
.
.
.
COMPLETE
```

The need to include comments in the body of the TREES was recognized and provided by assigning the "$" character as a comment symbol. This character may appear at any point in a line, either first as in FORTRAN followed by the comment or after a legal statement on the same line.

Another decision was made which would provide greater flexibility in the creation of rules for new types of functional groups. Rather than use a predefined list of functional group names which could be referred to in the language, we decided to allow the use of true variable names. In this way the set of functional groups may grow as new rules are discovered and added without any changes to the compiler or interpreter. This required the ability, in the compiler, to recognize and store the variable names. These 20-character names are then passed through to the interpreter program as a table and ultimately used in the output to the user.

Three final decisions were made to make the syntactic analysis of the language as easy as possible without greatly impacting its clarity. First, all words in the TREES must be separated by at least one blank. Second, it is a "reserved word" language, which means that no legal word in the language (e.g., "PEAK", "CONTINUE"; see grammar) may be used as a variable name. This restriction was not considered to be a serious problem in this case as the variables would be functional group names, and the probability of conflict is low. Third, no fixed organization of statements by line was required. That is, statements may begin at any point on a line and may continue on the next line at any point. There may also be two legal statements on a line. The only restriction is that no break may occur in a word. One result of this free format design is that no continuation character is needed.

The final step in this process was to produce a formal grammer from the statements and the other information. The grammar puts the logic of the language into a simple form which allows checking for consistency and provides a clear expression of the language for building the compiler. A Backus Nauer form[7] grammar may be found in the Appendix along with an explanation of its meaning.

As a second means of illustrating the nature of concise, a complete tree for the aldehyde rules is presented (Figure 1). This tree was taken from the set of working rules, with the only modification being the addition of indentation to show the nesting of the statements. The first line is the HEADER, which begins with the group name ALDEHYDE. This is

Table I. Parse Table for the Reduced Concise Language

	words								
state	BEGIN (1)	CMD (2)	DONE (3)	ELSE (4)	END (5)	FINISH (6)	IF (7)	STMT (8)	THEN (9)
1 (start)		$I = 20$ $E(L) = \text{NO}$ $S = 1$	$I = 33$ $E(L) = \text{NO}$ $L = L - 1, L \geq 1?$ $S = 1$	$I = 32$ $E(L) \equiv \text{YES}?$ $E(L) = \text{NO}$ $L = L + 1$ $S = 4$	$I = 42$ $L \equiv 1?$ $S = -1$	$I = 21$ $E(L) = \text{NO}$ $S = 1$	$I = 10$ $E(L) = \text{YES}$ $S = 2$		
2								$I = 0$ $S = 3$	
3				$I = 32$ $E(L) \equiv \text{YES}?$ $E(L) = \text{NO}$ $L = L + 1$ $S = 4$					$I = 31$ $L = L + 1$ $S = 4$
4	$I = 0$ $S = 5$	$I = 20$ $I = 33$ $L = L - 1, L \geq 1?$ $S = 1$				$I = 21$ $I = 33$ $L = L - 1, L \geq 1?$ $S = 1$			
5		$I = 20$ $S = 6$				$I = 21$ $S = 6$	$I = 10$ $E(L) = \text{YES}$ $S = 2$		
6		$I = 20$ $S = 6$	$I = 33$ $E(L) = \text{NO}$ $L = L - 1, L \geq 1?$ $S = 1$			$I = 21$ $S = 6$	$I = 10$ $E(L) = \text{YES}$ $S = 2$		
	initialize $L = 1$ $E(1\text{-}250) = \text{NO}$								

followed by the group empirical formula including the degree of unsaturation present. The last part is an index which the compiler uses to check that the trees are in correct order. These trees were numbered by starting at 100 in increments of 100 to allow room for insertions. Following the header, the body of the tree begins, here, with an "IF" statement refering to the presence of a carbonyl stretching peak. In this tree all other statements depend on the truth of this first statement, if it is false the next statement reached (at its own level of indentation) is the FINISH which stops further consideration of this tree. If the first statement is true, the "THEN BEGIN" block is entered, and another peak is checked for. If this is also true the aldehyde value is set to 0.25. At this point another group value (previously analyzed) is checked. If there is a high probability for the ACID group, the presence of the aldehyde is less easy to determine, so its own probability is reduced by 0.15. If ACID is below 0.90, consideration continues by asking if the empirical formula contains nitrogen and continues with questions about other peaks. Near the center of the tree, a statement refers to the solvent used. If it was run in oil, a peak of interest may be masked; if not, however, a peak statement can increase the aldehyde probability value. The final section is concerned with identifying the subgroup types, saturated or unsaturated aldehyde. Although other variations and types of statements are possible in CONCISE, this illustrates the way IR rules can be expressed in this language.

COMPILER

Having defined a language in which to encode the rules for IR interpretation, one requires a compiler to convert the "TEXT" rules into a more compact form which could be efficiently used by the main interpreter program. This compiler required three functions: first, to identify individual statements as they appear one word at a time in the input stream and to extract the relevant information from each statement; second, to analyze the IF–THEN–ELSE control logic; third, to assign numerical codes to the various pieces of information and write them in a form suitable for the planned interpreter program. Also implicit in this processing is error checking of the statements and the logic. The types of errors which could be identified are incorrect spelling, incorrect word order, incorrect nesting of "THEN" and "ELSE" clauses, and illegal variable names. Identification of an error causes an error statement to be printed and compiled code to be deleted. Compilation continues in order to identify any other errors in the input file.

The actual compiler program consists of three parts, the text input and identification (syntactic analysis), the parsing of the statements, and the code assignment and output. Input processing is simplified by the requirement of one or more blanks between input words. The input text is read one line at a time and each "WORD" is in turn isolated and passed to the syntactic analysis phase. The analysis consists of first comparing the "WORD" against a dictionary of words in the language (e.g., "PEAK"); if a match is found its number is returned. Otherwise, a test is made to see if the "WORD" is a number. It may be either an integer or a floating point number. If it is a number, an integer or floating point code is assigned, and the value is returned. Otherwise, the "WORD" is assumed to be a variable name. The "WORD" is compared against a list of variable names already found in the input. If a match is found, its code is returned. If it is not found, the "WORD" is added to the list, and its newly assigned code is returned. In this way an input "WORD" may be identified in one of five classes: a legal word in the language, an integer, a floating point number, an old variable name, and a new variable name, each with a class code and an individual code. The codes are then passed to the parser. The heart of this section is the parse table, an example of which is shown in Table I.

This table is a condensed version which deals only with the control logic, the most difficult part of the process. However, the operation is the same for the full table as built from the full language. Before operation of the parser can be explained, several points must be made. First, as a "THEN" or "ELSE" block is entered the level ("*L*") increases by 1. Second, there

Table II. Keyword Information and Codes

statement	keyword	code	saved information
QUESTIONS:	PEAK	11	the number of peaks specified and a separate code indicating an exact number of peaks or a minimum number of peaks or "ANY" number of peaks (not zero) which fit the following conditions: intensity range 1-10 (weak-strong), width range 1-3 (sharp-broad), position range 4000-550 cm^{-1}
	PROBABILITY	12	variable name comparison type (.GT., .EQ., .LT.) value or second variable name
	EMPIRICAL FORMULA	13	number of atoms atom type original or updated formula to be checked
	SOLVENT	14	solvent type
COMMANDS:	CONTINUE	21	
	FINISH	22	
	SET	23	variable name value or second variable name
	ADD SUBTRACT MULTIPLY DIVIDE	24	code (A, S, M, D) variable name value or second variable name
	UNKNOWN	25	variable name
	ADD SUBTRACT	26	empirical formula
CONTROL:	THEN	31	
	ELSE	32	
	DONE	33	
	HEADER	41	variable name empirical formula
	END	42	
	COMPLETE	43	

is an indicator ("*E*(*L*)") which tells if an "ELSE" clause may be accepted at any given point. The next state ("*S*") to be used is also given, along with the instruction ("*I*") code. Empty boxes indicate errors. All "IF" statements have been condensed to the general statement "STMT" and all commands to "CMD". For parsing the statement IF STMT THEN BEGIN CMD DONE ELSE CMD use the algorithm shown in Chart III.

There is a short FORTRAN program in the Appendix which implements this process. Although not shown in this table, in the full table there are also codes to indicate the beginning and end of all statements and which are the terms and numbers to save for ultimate output. When a term is found which indicates the end of a statement, all saved information is sent to the output routines. In this compiler the codes are packed into assigned bit fields to save more space. However, this packing step must be eliminated to implement the compiler on computers with other than 32-bit word lengths (e.g., the version implemented on an instrument-based minicomputer[1c]). Elimination of the packing step increased space requirements and running time only slightly.

Examples of the key information and codes used are summarized in Table II.

The final output from the compiler consists of two parts. The first is the list of all variable names found along with their numerical codes by which they are referred in the compiled output, and the second is the encoded rules. This output together with the spectral information serve as input to the interpreter program. One point to be made about the logical structure of the output instructions is that it is unchanged by the compiler from the input and is still represented by the parse table (Table I).

Chart III. Parse Table Algorithm

```
   START:  SET .STATE. TO 1
           SET .LEVEL. TO 1                                          ("L")
           SET .ELSEFLAG(1 TO 250). TO ILLEGAL                       ("E(L)") = "NO"

     TOP:  READ WORD FROM INPUT (.WORD.)
           COMPARE .WORD. WITH LIST OF LEGAL WORDS
           IF NOT FOUND, GO TO ERROR-1
           LOOK IN ENTRY OF PARSE TABLE WHERE .STATE. AND .WORD. CROSS
           IF THE ENTRY IS BLANK, GO TO ERROR-2
           * OUTPUT THE INDICATED CODES *                            ("I")
           CARRY OUT THE INDICATED INSTRUCTION
           INCREASE OR DECREASE .LEVEL. (L = L+1 or L = L-1, must not go below 1)
           SET .ELSEFLAG(.LEVEL.). TO LEGAL ("YES") OR ILLEGAL ("NO")
           TEST IF .ELSEFLAG(.LEVEL.). IS LEGAL ("YES"))
           SET .STATE. TO NEXT STATE ("S")
           IF .STATE. IS "-1" GO TO BOTTOM
           GO TO TOP

 ERROR-1:  REPORT ERROR AND EXIT

 ERROR-2:  REPORT ERROR AND EXIT

  BOTTOM:  COMPILATION COMPLETE.
```

Chart IV. IF-THEN-ELSE Algorithm

```
   START:  SET .STATE. TO ACTIVE
           SET .LEVEL. TO 1

  HEADER:  READ INSTRUCTION FROM FILE (.CODE.)
           IF .CODE. IS ∅, GO TO HEADER
           IF .CODE. IS BEGINNING OF NEW TREE, GO TO TOP
           IF .CODE. IS "COMPLETE" INSTRUCTION, GO TO BOTTOM
           GO TO ERROR-1

     TOP:  READ INSTRUCTION FROM FILE (.CODE.)
           IF .CODE. IS "END" INSTRUCTION, GO TO START
           IF .CODE. IS AN "IF-STATEMENT" INSTRUCTION, GO TO QUERY
           IF .CODE. IS A "COMMAND" INSTRUCTION, GO TO CMND
           IF .CODE. IS A "THEN" INSTRUCTION, GO TO THEN
           IF .CODE. IS A "ELSE" INSTRUCTION, GO TO ELSE
           IF .CODE. IS A "DONE" INSTRUCTION, GO TO DONE
           IF .CODE. IS A "FINISH" INSTRUCTION, GO TO FINISH
           GO TO ERROR-2

   QUERY:  IF .STATE. IS INACTIVE, GO TO TOP
           EVALUATE IF-STATEMENT
           SET .LOGIC. TO TRUE OR FALSE
           GO TO TOP

    CMND:  IF .STATE. IS INACTIVE, GO TO TOP
           CARRY OUT COMMAND
           GO TO TOP

  FINISH:  IF .STATE. IS INACTIVE, GO TO TOP

   AGAIN:  READ INSTRUCTIONS FROM FILE (.CODE.)
           IF .CODE. IS "END" INSTRUCTION, GO TO START
           GO TO AGAIN

    THEN:  INCREASE .LEVEL. BY 1
           PUSH .LOGIC. ONTO THE STACK
           PUSH .STATE. ONTO THE STACK
           IF .STATE. IS INACTIVE, GO TO TOP
           IF .LOGIC. IS TRUE, SET .STATE. TO ACTIVE
           IF .LOGIC. IS FALSE, SET .STATE. TO INACTIVE
           GO TO TOP

    ELSE:  INCREASE .LEVEL. BY 1
           PUSH .LOGIC. ONTO THE STACK
           PUSH .STATE. ONTO THE STACK
           IF .STATE. IS INACTIVE, GO TO TOP
           IF .LOGIC. IS TRUE, SET .STATE. TO INACTIVE
           IF .LOGIC. IS FALSE, SET .STATE. TO ACTIVE
           GO TO TOP

    DONE:  DECREASE .LEVEL. BY 1
           IF .LEVEL. IS LESS THAN 1, GO TO ERROR-3
           POP .STATE. OFF THE STACK
           POP .LOGIC. OFF OF THE STACK
           GO TO TOP

 ERROR-1:  REPORT ERROR AND EXIT

 ERROR-2:  REPORT ERROR AND EXIT

 ERROR-3:  REPORT ERROR AND EXIT

  BOTTOM:  ALGORITHM DONE CORRECTLY, REPORT RESULTS.

    NOTE:  "FINISH" is similar to a FORTRAN "RETURN" which indicates all

           following instructions up to the "END" should be skipped.

    NOTE:  When a header code is found, the first step is to check the empirical

           formula to find if the tree is to be used.

    NOTE:  This algorithm has been encoded in a FORTRAN program and is included

           in the Appendix.  This program will follow the instructions output by

           the PARSER program.  Together they represent the framework of this

           type of language.
```

INTERPRETER (PAIRS)

The purpose of this program is to read the encoded rules,

Chart V

```
$
$ THIS IS THE GRAMMER FOR THE I.R. ANALYSIS PROGRAM LANGUAGE.
$
$ FILE DEFINITION.
$
   START = { DTREE SP } 'COMPLETE' .
$
$ TREE DEFINITION.
$
   DTREE = TREENAME SP SUBSTRUCTURE SP INDEX SP { STATEMENT SP }
           'END' .
$
   TREENAME = VARIABLE
$
   SUBSTRUCTURE = { ATOMTYPE SP NUMBER SP } .
$
   INDEX = NUMBER .
$
$ STATEMENT DEFINITION.
$
   STATEMENT = COMMENT ! COMMAND ! QUERY .
$
$  COMMENT = '$' 'ARBITRARY COMMENT' .
$
   COMMAND = 'CONTINUE' ! 'FINISH' ! PROBADJUST ! EFORMADJ .
$
$ QUERY DEFINITION.
$
   QUERY = 'IF' SP SITUATION SP 'THEN' SP ACTION [ SP 'ELSE'
           SP ACTION ] .
$
$ QUESTION DEFINITION.
$
   SITUATION = PEAKSQ ! PROBQ ! EMPFORMQ ! SOLVQ .
$
   PEAKSQ = COUNT SP 'INTENSITY' SP ( NUMBER ! NUMBER SP TO SP
            NUMBER ) SP ( WIDTH ! WIDTH SP TO SP WIDTH )
            SP PEAKT SP BE SP 'IN' SP 'RANGE' SP NUMBER SP TO
            SP NUMBER .
$
   PROBQ = GROUPTYPE SP RELATION SP ITEM .
$
   EMPFORMQ = COUNT SP ATOMTYPE SP LEFT .
$
   SOLVQ = 'SPECTRUM' SP 'RUN' [ SP 'IN' ] SP SOLVTYPE .
$
$ ACTION DEFINITION.
$
   ACTION = COMMAND ! SP 'BEGIN' SP STATEMENT SP
            { STATEMENT SP } 'DONE' .
$
   PROBADJUST = ADJTYPE1 ! ADJTYPE2 ! ADJTYPE3 ! ADJTYPE4 .
$
   ADJTYPE1 = 'SET' SP GROUPTYPE SP TO SP ITEM .
$
   ADJTYPE2 = ARITH2 SP ITEM SP DIR SP GROUPTYPE .
$
   ADJTYPE3 = ARITH1 SP GROUPTYPE SP 'BY' SP ITEM .
$
   ADJTYPE4 = GROUPTYPE SP 'UNKNOWN' .
$
   EFORMADJ = ARITH2 SP SUBSTRUCTURE SP DIR SP 'FORMULA' .
$
$ MISC. DEFINITION.
$
   GROUPTYPE = VARIABLE .
$
$  VARIABLE EXAMPLE:
$                    METHYL
$                    METHYL-GEMDI
$                    NITRO
$                    NITRO-SATURATED
$
   ITEM = VALUE ! GROUPTYPE .
$
   NUMBER = DIGIT [ DIGIT ] .
$
   VALUE = DIGIT DOT [ DIGIT ] .
$
   DOT = '.' .
$
   DIGIT = '0' ! '1' ! '2' ! '3' ! '4' ! '5' ! '6' ! '7' ! '8' !
           '9' .
$
   ATOMTYPE = 'H'  ! 'B'  ! 'C'  ! 'N'  ! 'O'  ! 'F'  ! 'P'  !
              'S'  ! 'CL' ! 'BR' ! 'I'  ! 'UNSAT' ! 'NONH' .
$
   ARITH1 = 'MULT' ! 'DIV' .
$
   ARITH2 = 'ADD' ! 'SUBT' .
$
   DIR = TO ! 'FROM' .
$
   TO = 'TO' .
$
   COUNT = 'ANY' ! 'NO' ! 'AT' SP 'LEAST' NUMBER ! NUMBER .
$
   WIDTH = 'SHARP' ! 'AVERAGE' ! 'BROAD' .
$
   BE = 'IS' ! 'ARE' .
$
   LEFT = 'REMAIN' ! 'REMAINS' ! 'EXIST' ! 'EXISTS' .
$
   RELATION = 'GREATER' SP 'THAN' ! 'EQUAL' SP TO !
              'LESS' SP 'THAN' .
$
   SOLVTYPE = 'OIL' ! 'NEAT' ! 'NACL' ! 'CCL4' ! 'CHCL3' .
$
   PEAKT = 'PEAK' ! 'PEAKS' .
$
   SP = ' ' .
$
$  END END END END END END END END END END END END END END END
```

prompt the user for facts relating to the spectrum, and finally produce a probability for each group type referred to in the encoded rules. This discussion will first cover the structure of the program and then the method used to apply the rules to the spectrum.

The PAIRS program consists of three major sections: spectrum input, rule interpretation, and probability output. In the input section, information about a spectrum may either be read from a previously created "SPECTRUM" file or from the terminal. The interactive nature of this section has already been described;[1] however, an outline of the method will be given. If manual input is selected, the information requested is as follows: a title to identify spectrum; the solvent used; the empirical formula (if available); peak position, intensity, and width.

A final step in the input section is the option to review and to edit any and all input data. For example, if an error has been found, a new peak may be inserted or its position shifted. Also, if empirical formula information becomes available, it may be added or modified.

In all interactions involving user input, there is an effort to recognize and report any and all errors without system tracebacks and termination of the program (particularly where text is entered in place of a number). To provide this feature, all input is accepted as text, and numbers are recognized and converted by our own routines. This process also allows free format input of mixed text and numerical input on a line, which is not normally convenient in FORTRAN. Finally, this allows a "HELP" request to be accepted at any point in the input and correctly handled.

Once the input has been verified and accepted by the user, all of the spectral information is written into a file, and processing is passed to the second section, the interpretation procedure. In this section, which is run without user intervention, the compiled rules are read in sequentially and either evaluated or discarded. There are two major features of this process. The first is the method used to control evaluation of the rules, and the second is the actual method for evaluating the rules. The result of compilation is a sequence of codes for the various statements and commands and also for the "THEN", "ELSE", and "DONE" words. During the process of interpretation, every code must be read, one at a time; however, from the logical structure of the language it is clear that not every instruction will be used. Following an "IF" statement, only the "THEN" or "ELSE" clause will be used, and the other will be ignored. In most computer languages, as in the hardware of most computers, there is a facility to jump around instructions with a "GO TO" type statement. In this case, because of the design of this language, this could not be done, so another method was needed to select and use only the correct codes. The algorithm shown in Chart IV describes the method.

The advantage of this method is that only one instruction at a time need be read by the interpreter, eliminating the need to have all the instructions in memory at one time. The result is that any length file can be processed, allowing the rules to grow, so only time and not computer memory is the limiting factor.

The second feature of the interpretation process involves the method used to evaluate the IF-STATEMENTS and process the COMMANDS. Simply, this is done by providing a FORTRAN subroutine for each IF-STATEMENT and COMMAND in the IR language. When the interpreter decides it has read an IF-STATEMENT or COMMAND, it looks at the exact code for it and calls the appropriate routine. Each routine then picks up the remaining information carried by the IF-STATEMENT or COMMAND. For an IF-STATEMENT the routine then compares this information with the spectrum and a true or false value results. In a COMMAND routine the information from the instruction is used to adjust a probability or the empirical formula. During this process the probabilities for the functional groups can be adjusted and given a final value. When this section is complete, every functional group has an assigned probability or the indication "UNKNOWN". In the third section, the results are reported in two ways: sorted by probability in decreasing order and sorted alphabetically by group name. These sorted lists are put in an output file following a summary of the spectral information.

CONCLUSIONS

A number of advantages have been realized by following this approach. The most important, we believe, is that information about IR interpretation remains at a high level, in explicit form rather than being encoded in FORTRAN in a way which makes it difficult to extract as information. This approach has proved valuable for the continued evolution of the rules by our efforts and by others not directly involved in this project. The time and effort required to develop this language and programs have been recovered already in savings over what would have been required to develop and maintain a similar system with the rules embodied in FORTRAN. Execution time for compilation of 7800 lines of rules is about 8 min on a VAX 11/780, carried out as a batch process. Interpretation requires several seconds for a typical spectrum of 25-30 peaks, which is within the bounds considered acceptable for an interactive program.

It is hoped that this discussion will serve to stimulate others, who may develop information-intensive software, to consider creation of special purpose languages for building their systems.

The compiler, interpreter, and current set of IR rules are available for distribution from the Quantum Chemistry Program Exchange, Bloomington, IN.[8]

APPENDIX A

The BNF grammar for the CONCISE language includes the following special symbols: "$" indicates a comment—not part of the grammar, ' ' indicates a literal—a word in the language, [] indicates an option—usually a word which will make a statement read better but does not add to the information, ! indicates alternative choices of words or characters, = defines the symbol on the left as the phrase on the right, . ends a phrase, () encloses options in a statement, {} indicates multiple repeat allowed (e.g., 0, 1, or more).

To read this grammar begin at "START" (see Chart V). This is defined as 0, 1, or more "DTREE"s followed by the word "COMPLETE". A "DTREE" is defined next and represents a block of rules for a major group (e.g., ketone) and its subtypes (e.g., α-β unsaturated). A DTREE starts with a major group name, followed by its empirical formula and an index which is used by the compiler to test if the groups of rules are in order. The body of the rules are made up of one or more statements and is concluded with "END". Each of the terms not in single quotes is further defined in the grammar. All terms must ultimately stop at an expression in single quotes. N.B.: reading the grammar is not the recommended way to learn a language; it is for construction of the compiler.

REFERENCES AND NOTES

(1) (a) Woodruff, H. B.; Smith, G. M. *Anal. Chem.* **1980**, *52*, 2321. (b) Woodruff, H. B.; Smith, G. M. *Anal. Chim. Acta* **1981**, *133*, 545. (c) Tomellini, S. A.; Saperstein, D. D.; Stevenson, J. M.; Smith, G. M.; Woodruff, H. B.; Seelig, P. F. *Anal. Chem.* **1981**, *53*, 2367. (d) Woodruff, H. B. "Progress in Industrial Microbiology"; Bushell, M. E., Ed.; Elsevier: Amsterdam, 1983; Vol. 17, p 71.
(2) Munk, M. E.; Shelley, C. A.; Woodruff, H. B.; Trulson, M. O. *Fresenius' Z. Anal. Chem.* **1982**, *313*, 473.
(3) Corey, E. J.; Wipke, W. T.; Crammer, R. D.; Howe, J. W. *J. Am. Chem. Soc.* **1972**, *94*, 421.
(4) Wipke, W. T.; Gund, P. *J. Am. Chem. Soc.* **1974**, *96*, 229.
(5) Shortliffe, E. H.; Buchanan, B. G. *Math. Biosci.* **1975**, *23*, 351.
(6) Aho, A. V.; Ullman, J. D. "Principles of Compiler Design"; Addison Wesley: Reading, MA, 1977.
(7) Wirth, N. *Commun. ACM* **1977**, *20* (11), 822.
(8) Smith, G. M.; Woodruff, H. B. *QCPE* **1981**, *13*, 426.

NLM-CHEMSORT: An Algorithm and Computer Program for Sorting Chemical Names

JOAN BURNSIDE, PAUL N. CRAIG, and GERARD T. GUTHRIE*
National Library of Medicine, Bethesda, Maryland 20209

Received August 8, 1983

An algorithm is described that has been designed to sort medium-sized lists of chemical names, including common, generic, trivial, and systematic names and code numbers, into a logical sequence. It successfully sorted more than 99.5% of 3767 names in its first application. Minor revisions then resulted in more than 99.9% success with the same set of names. The algorithm generates an 80-character primary sort key (alphabetic characters only) and a 16-character secondary level sort key (alphanumeric characters). These sort keys are generated de novo from the name as needed and, thus, do not require increased permanent-storage costs. Sorting on the primary sort key (and secondary sort keys when identical primary keys exist) results in logical sequences of chemical names.

BACKGROUND

Since its inception in 1969, the Toxicology Information Program (TIP) at the National Library of Medicine (NLM) has built various on-line files, based on either bibliographic records or chemical substances.[1] For large files of chemicals such as CHEMLINE (>500000 compounds), there is no need for a printed list of names, since these records can be best accessed randomly by on-line searching. But for smaller on-line files such as the Toxicology Data Bank (TDB),[1] which contains detailed records for some 4000 compounds, there is a recurring need for printing lists by chemical names.

Until recently, TIP scientists relied on printed lists that were sorted by standard computer programs. With the initiation of a collaborative effort between NLM and the National Toxicology Program (NTP) in 1979, the need for sorted chemical name listings increased. When the standard computer sort routine was used on these lists (ranging from 100 to 5000 names), the resulting indexes often separated related