SRI International

Technical Note 138 • February 1977

THE LIFER MANUAL
A Guide to Building Practical Natural Language Interfaces

Prepared by:

Gary G. Hendrix
Artificial Intelligence Center
Computing and Engineering Sciences Division

The work reported herein was conducted under SRI's Internal Research and Development Program.

ABSTRACT

This document describes an application-oriented system for creating natural language interfaces between existing computer programs (such as data base management systems) and casual users. The system is easy to use and flexible, offering a range of capabilities that support both simple and complex interfaces. This range of capabilities allows beginning interface builders to rapidly define workable subsets of English and gives more advanced builders the tools needed to produce powerful and more efficient language definitions. The system includes an automatic mechanism for handling certain classes of elliptical (incomplete) inputs, a spelling corrector, a grammar editor, and a mechanism that allows even novices, through the use of paraphrase, to extend the language recognized by the system. Experience with the system has shown that for many applications, very practicable interfaces may be created in a few days.

CONTENTS

I. AN OVERVIEW OF LIFER

LIFER is a practical system for creating English language interfaces to other computer software (such as data base management systems and expert consultant programs). Its purpose is to make the competence of other computing systems more readily accessible by overcoming the language barriers separating these systems from potential users. Emphasizing human engineering, LIFER has bundled natural language specification and parsing technology into one tidy package, which includes an automatic facility for handling elliptical (i.e., incomplete) inputs, a spelling corrector, a grammar editor, and a mechanism that allows even novices, through the use of paraphrase, to extend the language recognized by the system. Offering a range of capabilities that supports both simple and complex interfaces, LIFER allows beginning interface builders to rapidly create workable systems and gives ambitious builders the tools needed to produce powerful and more efficient language definitions. Experience with LIFER has shown that for some applications, very comfortable interfaces may be created in a matter of days. The resulting systems are directly usable by such people as business executives, office workers, and military officials whose areas of expertise are outside the field of computer science.

Research in artificial intelligence and computational linguistics has not developed a general approach to the problems of understanding English and other natural languages. However, a number of mechanisms have been developed that make it possible to deal with major fragments of language pertinent to particular application areas. For many applications, an ability to communicate in terms of such fragments is both sufficient for the task at hand and clearly preferable to forcing users to learn an awkward, inflexible, error-intolerant, machine-oriented input format. LIFER's aim is to service such nearer-term, practical applications while the search for a unified and elegant theory of language understanding continues.

LIFER is composed of two basic parts: a set of interactive language specification functions and a parser. In standard practice, an interface builder uses the language specification functions to define an application language. This application language is a subset of a natural language (e.g., English) that is appropriate for interacting with an existing software product. Using this language specification, the LIFER parser can interpret natural language inputs, translating them into appropriate interactions with the application software.

Example interactions with a LIFER application language for a data base access system are presented in Figure 1. (This particular language definition, called INLAND, was developed by Earl Sacerdoti and others (Sacerdoti, 1977) as a part of SRI's LADDER project for ARPA.) The system user types in a query or command in ordinary English, followed by a carriage return. The LIFER parser then processes the input. When syntactic analysis is complete, LIFER types "PARSED!" and invokes application software to respond.

An important feature of the LIFER parser is its ability to process elliptical (incomplete) inputs. Thus, if the system has been asked

WHAT IS THE SPEED OF THE KITTY HAWK

the subsequent input

OF THE ETHAN ALLEN

will be interpreted as WHAT IS THE SPEED OF THE ETHAN ALLEN. Analysis of incomplete inputs is performed automatically by LIFER, making it unnecessary for the interface builder to explicitly define elliptical constructions in the application language.

If a user misspells a word, LIFER attempts to correct the error using the INTERLISP spelling corrector (Teitelman, 1975). If the parser cannot account for an input in terms of the application language definition, user-oriented error messages are printed that indicate what LIFER was able to understand and that suggest means of correcting the error. (See example.)

```
-What is the speed of the Kitty Hawk
PARSED!
((SPEED 35 KNOTS))

-Of the Ethan Allen
TRYING ELLIPSIS:  WHAT IS THE SPEED OF THE ETHAN ALLEN
((SPEED 30 KNOTS))

-Displacement
TRYING ELLIPSIS:  WHAT IS THE DISPLACEMENT OF THE ETHAN ALLEN
((STANDARD-DISPLACEMENT 6900 HUNDRED-TONS))

-length of the fastest Soviet sub
TRYING ELLIPSIS:  WHAT IS THE LENGTH OF THE FASTEST SOVIET SUB
((LENGTH 285 FEET / SPEED 30 KNOTS))

-who onws the KIEV
     OWNS <==(assumed spelling error)
PARSED!
((COUNTRY USSR))

-who owns the JFK
TRYING ELLIPSIS:  ELLIPSIS HAS FAILED
THE PARSER DOES NOT EXPECT THE WORD "JFK" TO FOLLOW "WHO OWNS
THE"
OPTIONS FOR NEXT WORD OR META-SYMBOL ARE:
<SHIP-NAME>

-Define JFK to be like Kennedy
PARSED!
 . {JFK is now a synonym for KENNEDY, which is a ship name}
 .
-REDO -2            {that is, parse WHO OWNS THE JFK}
PARSED!
((COUNTRY USA))

-? BUILT LAFAYETTE
TRYING ELLIPSIS:  ELLIPSIS HAS FAILED
 . {error message omitted}
 .
-Let "? built Lafayette" be a paraphrase of "who built the
Lafayette"
PARSED!
 .
 .
-? built Lafayette
PARSED!
((BUILDER GENERAL.DYNAMICS))

-owns longest nuclear submarine
TRYING ELLIPSIS:  ? OWNS LONGEST NUCLEAR SUBMARINE
((COUNTRY USSR / LENGTH 426 FEET))
```

Although the language specification task logically precedes the processing of inputs by the parser, the two may actually proceed in parallel. That is, the application language need not be completely specified in advance of any parsing. Rather, some types of inputs may be defined first, and the parser used on them. Later, the language specification may be extended interactively. This ability to intermix parsing and language specification activities allows interface builders to function in a rapid, extend-and-test mode. The immediate feedback produced by this mode of operation is an important factor in reducing the time required to construct interfaces. Looked at another way, it extends the richness of the language fragment that can be developed in a given length of time.

An interesting and important ramification of the intermixing of language specification and parsing operations is that it is possible to bootstrap to the language specification functions themselves. By defining an interface to LIFER's own language specification functions (particularly the function PARAPHRASE), it becomes possible for naive users to give natural language commands for extending the language. This is illustrated by the paraphrase example of Figure 1.

The LIFER parser uses an augmented, finite state transition network (Woods, 1970). The LIFER language specification functions construct these underlying transition networks automatically from language production rules of the type commonly used by both natural linguists and compiler builders.

The production rules may be modified easily and tested interactively, allowing sophisticated language definitions to be produced within a short period of time.

LIFER is currently coded in INTERLISP on the PDP-10. This makes interfaces to other INTERLISP programs most convenient, but interfaces to programs written in other languages have been built. In fact, the LADDER system of SRI uses LIFER to interface over the Arpanet with remote computers whose local programs accept only DATALANGUAGE.

In addition to parsing and language specification functions, the LIFER package also includes a comprehensive set of utility routines for interrogating and editing the application language, and for compiling and saving language specifications on files.

II. THE LIFER APPROACH TO LANGUAGE

The LIFER package includes neither a grammar nor a semantics for any language. Rather, it contains a set of interactive functions that facilitate the grammatical specification of a language fragment that is oriented toward the interface builder's specific application. The semantics of this language specification is typically carried by the existing programs to which the interface builder wishes to add a natural language front end.

Each call to one of the LIFER language specification functions causes internal structures to be built for subsequent use by the LIFER parser. Typically, many of the specification calls will indicate associations between certain linguistic constructions and the application software. The principle internal structures that are produced by the language specification functions are transition trees, which are a simplification of the augmented transition networks of Woods (1970). Using the transition trees and other internal structures, the parser interprets inputs in the application language. As a result of such interpretations, certain routines specified by the interface builder are invoked. It is through these invocations that the back-end application programs are activated.

In using LIFER, interface builders typically (but not necessairly!) embed considerable semantic information in the the syntax of the application language. For example, words like JOHN and AGE would not be grouped together into a single <NOUN> category. Rather, JOHN would be treated as a <PERSON>, and AGE as an <ATTRIBUTE>. Similarly, very specific sentence patterns such as

WHAT IS THE <ATTRIBUTE> OF <PERSON>

are typically used in LIFER instead of more general patterns such as

<NOUN-PHASE> <VERB-PHRASE>.

For each syntactic pattern, the interface builder supplies an expression for computing the interpretation of instances of the pattern. Expressions for sentence-level patterns usually invoke application software to answer questions or carry out commands.

An example sequence of interactions defining a LIFER application language is shown in Figure 2. At this stage, the reader should not attempt to understand the various calls to LIFER language specification functions. These will be explained in subsequent chapters. The purpose of the example is simply to give the flavor of the LIFER approach to language specification.

Working through the example from the top, application information concerning biographic data for JEWELL.FLEMING and IVAN.FRYMIRE is first stored on property lists for later querying. Then function MAKE.SET is called to define some word/phrase categories. The category <ATTRIBUTE>, for instance, is defined to include such words as AGE and OCCUPATION. Next, function PATTERN.DEFINE is used to add the productions

```
    <ATTR-SET> => (<ATTRIBUTE>)
and <ATTR-SET> => (<ATTRIBUTE> AND <ATTR-SET>)
```

to the language definition, establishing an <ATTR-SET> as a sequence of one or more attributes separated by ANDs. The third call to PATTERN.DEFINE sets up a top-level sentence pattern of the form

```
WHAT <IS/ARE> THE <ATTR-SET> OF <PERSON>
```

which can match such queries as

```
WHAT IS THE AGE AND OCCUPATION OF JEWELL.FLEMING
```

The expression for computing the value of this query maps down the list of sought-after attributes and extracts their values from the property list of the <PERSON>. (For this example, the "application software" is the set of LISP property-list functions.)

After calling the function LIFER.INPUT, all lines of input are sent to the LIFER parser for processing. The first query of the example is a complete sentence, but the second is elliptical. Note that no special patterns were needed to deal with this elliptic query. A more complex use of MAKE.SET and examples of the spelling corrector are shown in later interactions.

FIGURE 2   DEFINING AN APPLICATION LANGUAGE

```
   {set up data to be queried}
-SETPROPLIST(JEWELL.FLEMING (AGE 35 OCCUPATION TEACHER HEIGHT 5.5
                             WEIGHT 105))
-SETPROPLIST(IVAN.FRYMIRE   (AGE 40 OCCUPATION FARMER HEIGHT 6.2
                             WEIGHT 225))

   {MAKE.SET and PATTERN.DEFINE extend the language definition}
-MAKE.SET(<PERSON> (JEWELL.FLEMING IVAN.FRYMIRE ...))
-MAKE.SET(<ATTRIBUTE> (AGE OCCUPATION HEIGHT WEIGHT))
-MAKE.SET(<IS/ARE> (IS ARE))
-PATTERN.DEFINE(<ATTR-SET> (<ATTRIBUTE>)
                (LIST <ATTRIBUTE>))
-PATTERN.DEFINE(<ATTR-SET> (<ATTRIBUTE> AND <ATTR-SET>)
                (CONS <ATTRIBUTE> <ATTR-SET>))
-PATTERN.DEFINE((WHAT <IS/ARE> THE <ATTR-SET> OF <PERSON>)
                (MAPCONC <ATTR-SET> (FUNCTION (LAMBDA (A)
                    (LIST A (GETPROP <PERSON> A))))))

   {a call to LIFER.INPUT sends subsequent inputs to the parser}
-(LIFER.INPUT)

   {start NL interactions using grammar defined above}
-what is the occupation of jewell.fleming
PARSED!
(OCCUPATION TEACHER)
-age and weight
TRYING ELLIPSIS:  WHAT IS THE AGE AND WEIGHT OF JEWELL.FLEMING
(AGE 35 WEIGHT 105)

   {MAKE.SET is called to add variety to persons' names}
   {leading ! sends line to LISP'S EVAL, instead of to parser}
-!MAKE.SET(<PERSON> ((JEWELL . JEWELL.FLEMING)
                     (IVAN . IVAN.FRYMIRE)
                     ((JEWELL FLEMING) . JEWELL.FLEMING)
                     ((IVAN FRYMIRE) . IVAN.FRYMIRE))

   {now more English input}
-what is the height of ivan frymier
 (assumed spelling error)==>FRYMIRE
PARSED!
(HEIGHT 6.2)
-of jewell
TRYING ELLIPSIS: WHAT IS THE HEIGHT OF JEWELL
(HEIGHT 5.5)
   {define a paraphrase in English}
-define "give the height of ivan" like "what is the height of ivan"
PARSED!
LIFER.TOP.GRAM => GIVE THE <ATTR-SET> OF <PERSON>
   {output above shows LIFER's generalization of the paraphrase}
   {now try an input based on the paraphrase above}
-give the age and occupation of jewell fleming
PARSED!
(AGE 35 OCCUPATION TEACHER)
```

III. SPECIFYING A LANGUAGE DEFINITION

The LIFER system contains numerous functions for specifying components of a language. In this section, these functions are presented in approximately their order of complexity. Some applications may require only a few fixed inputs. Others may best be served by the use of complex subgrammars to compute intermediate results. LIFER allows interface builders to pick only those features that meet their needs and the level of linguistic sophistication required.

In the discussion that follows, a knowledge of INTERLISP (Teitelman, 1975) will be assumed.

A. Fixed Patterns

1. Invariant Input -- Invariant Response

The essence of the LIFER approach to language specification is to allow the user to define a set of input patterns and their associated responses. LIFER then factors the patterns into efficient transition trees for use by the parser. Patterns may be given to the system by calling the function PATTERN.DEFINE. PATTERN.DEFINE may be called either by its full name or by its "nickname" PD. In its simplest usage, PD is a function of two arguments: a pattern and a response expression. A pattern is a list of symbols. A sequence of words matching the sequence of symbols on the pattern list is to be accepted as a sentence in the input language. The associated response expression may be any evaluable LISP S-expression. When a given pattern is recognized by the parser, the associated response expression is evaluated to produce the response to the input. For most applications, the response expression will usually be a call to the underlying software package to which LIFER is providing an interface. (If the response expression returns the special atom *ERROR*, the input is rejected on semantic grounds, and the parser looks for an alternative syntactic analysis.)

As a simple example, suppose the system is to respond to the input

THANK YOU

with the response

YOU'RE WELCOME

The pattern to be recognized is

(THANK YOU)

and one possible response expression is

```
(QUOTE (YOU'RE WELCOME))
```

Thus, at the top-level of INTERLISP, the call to PD is

```
PD ((THANK YOU) '(YOU'RE WELCOME))
```

or, if embedded in a larger S-expression,

```
(PD (QUOTE (THANK YOU)) (QUOTE (QUOTE (YOU'RE WELCOME))))
```

As soon as this call to PD has been processed, the parser will recognize the pattern as a legal sentence in the input language. The parsing of the new pattern may be tested by typing

```
; THANK YOU
```

when INTERLISP types its prompt character. LIFER spots the initial semicolon before INTERLISP can process the input line in the normal way. (This is accomplished through INTERLISP's LISPXUSERFN feature. See Section VII for other initial control characters.) Rather than the line going to EVAL or APPLY for normal LISP processing, everything to the right of the semicolon is processed by the parser. For the example at hand, the input matches the pattern (THANK YOU) and causes the expression (QUOTE (YOU'RE WELCOME)) to be evaluated. The result of this evaluation is then printed as a response to the input.

2. Invariant Input -- Variant Response

Even with fixed-input patterns, some interesting questions may be asked. One of these is

```
WHAT TIME IS IT
```

Suppose, GETTIME is the name of a function of no arguments that obtains the current time of day by consulting the computer's clock.* Then the pattern (WHAT TIME IS IT) may be defined with an appropriate response by the call

```
PD((WHAT TIME IS IT) (GETTIME))
```

----

*For TENEX INTERLISP, the body of function GETTIME might be (SUBSTRING (DATE) 11 18)

B. Meta Symbols

Even with variable responses, an application language would be very limited if all inputs had to correspond on a word for word basis with one of the patterns. To achieve greater flexibility, variables that may match any number of words or phrases may be included in patterns. Such variables, called "meta symbols", may appear both in patterns, where they are bound, and in response expressions, where their values influence computations.

As an example of the use of meta symbols, suppose the symbol

<PERSON>

is used (by means described shortly) to stand for any of the words

JOHN, TOM, MARY, SUE

and the pattern

(WHO IS THE FATHER OF <PERSON>)

is included in the language definition. Then such inputs as

WHO IS THE FATHER OF TOM

and

WHO IS THE FATHER OF MARY

will be recognized by the parser. The response expression that answers these input queries will make use of the binding of the variable <PERSON> in its computations.

Each of the several methods for defining meta symbols provides two pieces of information: specifications for what words or phrases may be matched by the meta symbol; and specifications (often implicit) for assigning values to the meta symbol based on the particular match.

For example, <PERSON> may be defined in such a way that if <PERSON> matches JOHN, then the variable <PERSON> becomes bound to the atom JOHN. This variable binding is then available for use in response expressions for patterns that contain <PERSON>. To see the use of <PERSON> in a response expression, assume that atoms (e.g., JOHN) that name persons have the property FATHER on their property lists. Then the response expression for

```
(WHO IS THE FATHER OF <PERSON>)
```

might be

```
(GETP <PERSON> 'FATHER)
```

This pattern and associated response expression may be defined in the application language by the function call

```
PD((WHO IS THE FATHER OF <PERSON>)
   (GETP <PERSON> 'FATHER))
```

To simplify discussion, the examples of response functions given in this manual will appeal to property lists. However, it should be understood that application programs may be called just as easily. Suppose, for example, that the interface builder has a function FOOFUN for computing the FOO of <X> and <Y> (e.g., the price of x delivered to y; the sum of x and y; the distance between x and y; the children of x and y; etc.).

Then a call to PD of the form

```
    (PD '(WHAT IS THE FOO OF <X> AND <Y>)
        '(FOOFUN <X> <Y>))
```

will cause responses to the inputs matching the pattern to be computed by applying function FOOFUN to the values of the meta symbols <X> and <Y>.

This very common type of pattern may be generalized to the more powerful

```
        (WHAT IS THE <FOO> OF <X> AND <Y>)
```

where <FOO> is a meta symbol that becomes bound to a function name and

```
            (APPLY* <FOO> <X> <Y>)
```

is used as the response expression. For example, in

```
                WHAT IS THE SUM OF 2 AND 3
```

<FOO> would match SUM and become bound to the LISP function name PLUS.

Although it is a good idea to name meta symbols in some distinguished way (such as using "<" and ">" delimiters), LIFER will accept any literal atom as a meta symbol.*

---

*An atom used as a meta symbol may also act as a word in the application language, but only if it is recognized by using the predicate mechanism discussed below.

C. Meta Symbols as Sets of Words and Phrases

One of the ways to define a meta symbol is to allow it to take any value from an explicit set of atoms. This is accomplished through a call to MAKE.SET of the form

```
(MAKE.SET symbol set-specification)
```

where symbol is the meta symbol being defined and set-specification is a list of atoms (and, as will be seen shortly, more complex S-expressions) that may match symbol. For example, the call

```
(MAKE.SET '<PERSON> '(JOHN TOM MARY SUE))
```

defines <PERSON> to be a meta symbol that stands for any member of the set {JOHN, TOM, MARY, SUE}. If the parser matches <PERSON> to an atomic member of this set-specification list, then that member becomes the value of the variable <PERSON>.

Sometimes it is inconvenient for the atom that matches a meta symbol to become the symbol's value. For example, suppose <ADJ> is to be taken from the set {TALL, HEAVY, OLD} and a pattern of the form

```
(HOW <ADJ> IS <PERSON>)
```

is to be defined. If atoms matching <PERSON> (such as JOHN) have properties on their property lists such as HEIGHT, WEIGHT and AGE, it would be convenient for <ADJ> to match the words TALL, HEAVY and OLD but take as its values the atoms HEIGHT, WEIGHT and AGE.

To accomplish this end, MAKE.SET allows the list that is its second argument to include both atoms and dotted pairs. Each atom on the list will both match the meta symbol and become bound as the symbol's value. If a pair appears in the list, the CAR of the pair will match the symbol but the CDR will be taken as the associated value.

Thus, <ADJ> may be defined by

```
     (MAKE.SET '<ADJ>
               '((TALL .  HEIGHT)
                 (HEAVY .  WEIGHT)
                 (OLD .  AGE)))
```

and the new input pattern may be defined by

```
     (PD '(HOW <ADJ> IS <PERSON>)
          '(GETP <PERSON> <ADJ>))
```

It is important to note that the CDR of a pair need not be an atom. For example, some comparative adjectives might be defined by

```
   (MAKE.SET '<CADJ>
             '((TALLER HEIGHT GREATERP)
               (SHORTER HEIGHT LESSP)
               (OLDER AGE GREATERP)
               (YOUNGER AGE LESSP)))
```

where the CDR of each pair is a list whose CAR is an attribute name and whose CADR is the name of an ordering predicate.

Suppose meta symbol <CADJ> is to be used in processing inputs such as

IS JOHN TALLER THAN SUE

To handle such inputs in a general way, a pattern along the line of

(IS <PERSON> <CADJ> THAN <PERSON>)

seems suitable. However, the suggested pattern includes two instances of the meta symbol <PERSON>. Although the parser will accept such patterns, the interface builder must be aware that when variable <PERSON> is bound for the second time, the first binding will be lost. To circumvent this problem, let <PERSON1> and <PERSON2> have the same definition that <PERSON> had before. This allows members of the same set to be matched twice in the same pattern while binding the results of the two matches to two separate variables.*

Using <PERSON1>, <PERSON2> and <CADJ>, a general pattern for accepting inputs such as

IS JOHN TALLER THAN SUE

may be set up by

----
*The best way to set up <PERSON1> and <PERSON2> given <PERSON> is by using subgrammar definitions such as (PD '(<PERSON>) '<PERSON> '<PERSON1>). See later comments about subgrammars.

```
      (PD '(IS <PERSON1> <CADJ> THAN <PERSON2>)
          '(APPLY* (CADR <CADJ>)
                   (GETP <PERSON1>
                         (CAR <CADJ>))
                   (GETP <PERSON2>
                         (CAR <CADJ>))))
```

The input "IS JOHN TALLER THAN SUE" will then ultimately be answered by what amounts to

```
      (GREATERP (GETP 'JOHN 'HEIGHT) (GETP 'SUE 'HEIGHT))
```

For convenience in coding, MAKE.SET may be called by the nickname MS. If MS is called twice with the same first argument (symbol) but different second argument (set-specification), then the symbol becomes defined over the union of the sets. If a word or phrase is twice indicated to belong to a given symbol's set, no action is taken unless the second definition conflicts with the first. (This circumstance is brought about by the use of pairs, e.g., set = ((FAST . SPEED) (FAST . VELOCITY) (LONG . LENGTH) ...).) Conflicts produce error messages, and the most recent definition overrides all others.

MAKE.SET may also be used to define a meta symbol in terms of fixed phrases. If an element of the set-specification has a CAR that is a list, then the meta symbol will match that list as a complete phrase. For example,

```
        (MS '<PERSON>
            '(((TOM SMITH) . TSMITH)
              ((JOHN DOE) . JDOE)))
```

will extend the definition of <PERSON> to include the phrases "TOM SMITH" and "JOHN DOE." If <PERSON> matches "JOHN DOE," then the variable <PERSON> will take the atom JDOE as its value.

D. Meta Symbols as Predicates

A second way of defining meta symbols allows the symbol to match any S-expression (atom, string, or list) that satisfies some predicate. This is accomplished by a call to MAKE.PREDICATE (nicknamed MP) of the form

```
(MAKE.PREDICATE symbol predicate)
```

where predicate is a LISP function of one argument. After such a call, the symbol will match any S-expression for which the application of the predicate returns a non-NIL value. When a match occurs using the predicate mechanism, the symbol takes as its value the (necessarily non-NIL) quantity returned by the application of the predicate.

One of the most important uses of predicates is in processing numbers, which cannot feasibly be enumerated in an explicit set. A meta symbol for an arbitrary number may be defined by

```
(MAKE.PREDICATE '<N1> 'NUMBERP)
```

To avoid having the same symbol appear twice in the same pattern, it may be necessary to define an <N2>, and so on.

Given appropriate definitions for <N1> and <N2>, the call

```
    (PD '(WHAT IS THE SUM OF <N1> AND <N2>)
        '(PLUS <N1> <N2>))
```

will set up the necessary internal structures to allow LIFER to respond to

```
 ;  WHAT IS THE SUM OF 123 AND 456
```

with

```
 579.
```

Much of the power of the predicate feature comes from the ability to tear atoms apart by using UNPACK. Through this means, coded names (such as part designations, ID numbers, and the like) may be broken apart and analyzed. For example, the chemical formulas H20 and C2H5OH may be broken up into the lists (H 2 O) and (C 2 H 5 O H) for further processing. This processing (including the rejection of words that ought not to match the meta symbol) may be done by the interface builder's specialist routines. As one option, the interface builder may make a subordinate call to the LIFER parser with a grammar especially designed to interpret or reject sequences of characters constituting special coded symbols. (See discussion of function SUBPARSE below.)

E. Meta Symbols as Subgrammars

A third method for defining a meta symbol allows the symbol to match phrases that are defined in terms of patterns such as those discussed previously for defining sentence-level structures. The patterns used to define a meta symbol may themselves contain meta symbols, including the symbol being defined.

Each pattern used in the definition of a meta symbol is associated with a response expression, such as those described earlier. However, the value computed by this response expression is not printed as a top-level response, but becomes the value of the defined meta symbol. By this means, the value is available for use in "higher level" response expressions in which the meta symbol is referenced. This notion will be clarified shortly by examples.

1. Using PD to Define Subgrammars

The function PD (or PATTERN.DEFINE) was discussed above as a device for specifying sentence-level patterns, but it may be used to associate patterns with meta symbols also. PD is actually a function of three arguments. (In previous examples, the third argument has implicitly been NIL). If PD is called with a meta symbol (i.e., non-NIL atom) in either the first or third position of the argument list, then the pattern and response expression have no direct effect on what constitutes a complete sentence in the application language. Rather, the pattern and response expression become part of the definition of the meta symbol.

For example, the following two calls to PD both use meta symbol <ADDRESS> as a third argument. In the call at the left, the meta symbol is in the first position of the list of arguments. In the call on the right, it is in the last position. The calls are equivalent.

```
  PD(<ADDRESS>                              PD((<N1> <STREET-NAME>)
     (<N1> <STREET-NAME>)                      (LIST <N1> <STREET-NAME>)
     (LIST <N1> <STREET-NAME>))                <ADDRESS>)
```

Both calls allow symbol <ADDRESS> to match the pattern

(<N1> <STREET-NAME>)

Hence, <ADDRESS> can match such phrases as

333 RAVENSWOOD

909 BROADWAY

When such a match is made, variable <ADDRESS> becomes bound to a list such as (333 RAVENSWOOD). This value can be used in computing responses to sentence-level inputs following such patterns as

(WHAT BUSINESS IS LOCATED AT <ADDRESS>)

2. Recursive Subgrammars

Suppose there is a need to recognize phrases like

MARY AND SUE AND TOM

which join an arbitrary number of names with ANDs. This may be done by defining a recursive subgrammar as follows. First, a call to PD is used to set up a meta symbol called <PEOPLE>, which will be used to combine one or more instances of <PERSON>.

```
        PD((<PERSON>)
           (LIST <PERSON>)
           <PEOPLE>)
```

This call creates structures allowing the meta symbol <PEOPLE> to match the pattern

```
                            (<PERSON>).
```

Thus,

```
                                JOHN
```

(which is a <PERSON>) matches <PEOPLE>. The variable <PEOPLE> is bound to the value returned by the response expression. So, if <PEOPLE> matches JOHN, <PEOPLE> is bound to the list (JOHN).

Using a second call to PD, the patterns that specify <PEOPLE> are extended.

```
        PD((<PERSON> AND <PEOPLE>)
           (CONS <PERSON> <PEOPLE>)
           <PEOPLE>)
```

After this call, <PEOPLE> will match patterns of the form

```
                     (<PERSON> AND <PEOPLE>)
```

as well as patterns of the form

```
                            (<PERSON>)
```

Thus, sometimes through recursive applications of its definition, each of the following will be matched by <PEOPLE>.

```
                JOHN
                TOM AND JOHN
                SUE AND TOM AND JOHN
                MARY AND SUE AND TOM AND JOHN
```

The patterns and response expressions for <PEOPLE> have been defined in such a way that <PEOPLE> is always bound to a list of individuals such as:

```
(JOHN)
(TOM JOHN)
(SUE TOM JOHN)
(MARY SUE TOM JOHN)
```

To see how the two patterns for <PEOPLE> work together, consider the recognition of the phrase

TOM AND JOHN.

This phrase will match <PEOPLE> using pattern

(<PERSON> AND <PEOPLE>)

if TOM matches <PERSON>, and <PEOPLE> matches JOHN.

The phrase

JOHN

will match <PEOPLE> using the pattern

(<PERSON>)

if JOHN matches <PERSON>. Since JOHN does match <PERSON> with <PERSON> = JOHN, JOHN matches <PEOPLE> with

```
<PEOPLE> = (LIST 'JOHN)
         = '(JOHN)
```

Because JOHN matches <PEOPLE> with <PEOPLE> = '(JOHN) and TOM matches <PERSON> with PERSON = TOM, TOM AND JOHN matches <PEOPLE> with

```
                <PEOPLE> = (CONS 'TOM '(JOHN))
                         = '(TOM JOHN)
```

Once a subgrammar meta symbol such as <PEOPLE> has been defined, it may be used in other patterns. Indeed, <PEOPLE> was used above in the definition of itself. Consider now the use of <PEOPLE> in establishing a pattern for sentences such as

```
           JOHN AND TOM WORK IN DEPARTMENT A
           SUE AND MARY AND GENIE WORK IN DEPARTMENT B
```

The following PD call might be used to allow these inputs:

```
        PD((<PEOPLE> WORK IN DEPARTMENT <DEPT.NAME>)
           (PROGN
                (MAPC <PEOPLE>
                        (FUNCTION (LAMBDA (P)
                                (PUT P 'DEPARTMENT
                                       <DEPT.NAME>))))
                (QUOTE (I UNDERSTAND))))
```

where <DEPT.NAME> is a meta symbol, which matches department names. The response expression maps down the list of people making <DEPT.NAME> the value of the DEPARTMENT property of each person on the list. After completing the map, the response expression returns the message

```
                        (I UNDERSTAND)
```

for output by the system.

3. Multiple Word Names

One use of subgrammars that is of importance in many applications is the recognition of names composed of multiple words such as

SAM HOUSTON
GENERAL ELECTRIC
DODGE DART SWINGER
SAN MATEO COUNTY

These names may be recognized as phrases and associated with single-atom internal names (such as GE for GENERAL ELECTRIC) by a subgrammar. Such a subgrammar could be set up by

```
   (MAPC '(((SAM HOUSTON) . HOUSTON)
           ((GENERAL ELECTRIC) . GE)
           ((DODGE DART SWINGER) . DART-SWG)
           ((SAN MATEO COUNTY) . SMATCO))
         (FUNCTION (LAMBDA (N)
           (PD (CAR N) (LIST 'QUOTE (CDR N)) '<NAME>) )) )
```

Equivalent internal structures may be created by MAKE.SET, using the call

```
   (MAKE.SET '<NAME>
             '(((SAM HOUSTON) . HOUSTON)
               ((GENERAL  ELECTRIC) . GE)
               ((DODGE DART SWINGER) . DART-SWG)
               ((SAN MATEO COUNTY) . SMATCO)))
```

F. Additional Information about PATTERN.DEFINE

1. Semantic Tests and Context Sensitivity

a. The *ERROR* Feature

For some applications, it is convenient (or even necessary) to allow phrases to be rejected on semantic grounds, even when they are syntactically correct. For example, a sentence might be defined by the pattern

(<SUBJECT> <PREDICATE>)

with the restriction that the <SUBJECT> must be "appropriate" for the <PREDICATE>. For the <PREDICATE> "IS THE FATHER OF JOHN", the <SUBJECT> "WHO" or "SAM" is appropriate; but the <SUBJECT> "SUE", or "THE TABLE" is not.

The response expression associated with a pattern may perform tests to determine whether a particular combination of bindings for the pattern variables makes sense. If the bindings do make sense in combination, then a composite value should be returned as usual. However, if the test fails, the response expression should return the special atom *ERROR*. The LIFER parser will detect this error condition and reject the phrase combination. Such semantic-oriented phrase rejections may be made at both the sentence level and in subgrammars. After a failure, the parser continues to look for alternative syntactic analyses, just as if the failure were due to syntax.

In the <SUBJECT>-<PREDICATE> example, assume that the value of both the <SUBJECT> and the <PREDICATE> is a list of properties. For the <PREDICATE> "IS THE FATHER OF JOHN", the property list might be

```
(RELATION FATHER-OF
 OBJECT (FIRST-NAME JOHN LAST-NAME SMITH)
 +SUBJECT-RESTRICTIONS (PERSON)
 -SUBJECT-RESTRICTIONS (FEMALE CHILD))
```

Possible candidates for <SUBJECT> and their values are

```
WHO
     (FEATURES (QUERY ANIMATE PERSON))

SAM SMITH
     (FEATURES (PERSON ANIMATE MALE ADULT)
      FIRST-NAME SAM
      LAST-NAME SMITH)

SUE
     (FEATURES (PERSON FEMALE CHILD)
      FIRST-NAME SUE)

THE TABLE
     (FEATURES (INANIMATE FURNITURE SURFACE)
      LOCATION DINING-ROOM)
```

A response expression for the (<SUBJECT> <PREDICATE>) pattern might behave like this: Before combining a <SUBJECT> with a <PREDICATE>, the response expression checks whether all features listed in the +SUBJECT-RESTRICTIONS of the <PREDICATE> are included in the FEATURES list of the <SUBJECT>. The <SUBJECT> must have every one of these positive features if it is to be combined with the <PREDICATE>. A further test is made to determine if any of the -SUBJECT-RESTRICTIONS of the <PREDICATE> are included on the FEATURES list of the <SUBJECT>. The <SUBJECT> must have none of these negative features if it is to be combined with the <PREDICATE>. If these tests are passed, an appropriate response is computed. If the tests fail, the atom *ERROR* is returned.

b. Tradeoffs

There exist languages (e.g., context-sensitive languages) that necessarily require the use of the *ERROR* feature (or something like it) for their recognition. However, in building practical systems, the *ERROR* feature is seldom really required. The piece of language specified by a pattern of the form

(<X> <Y>)

that requires semantic testing can usually be specified alternatively by a sequence of patterns

```
(<X1> <Y1>)
(<X2> <Y2>)
.
.
(<Xn> <Yn>).
```

The various <Xi> and <Yi> match subsets of the phrases matched by <X> and <Y>, respectively. Further, for each i, the <Xi> and <Yi> are so constructed that any phrase matching <Xi> will be compatible with any phrase matching <Yi> in the pattern

```
(<Xi> <Yi>).
```

Thus, the syntactic restrictions on <Xi> and <Yi> eliminate the need for compatibility tests in defining the unifying pattern.

In general, if the splitting of <X> and <Y> produces a relatively small number of new patterns and meta symbols, and if most of the new meta symbols have a "natural" semantic interpretation, then the *ERROR* feature should not be used. The reasoning behind this rule of thumb is simple: LIFER is syntax oriented. By recording semantic distinctions in the syntax rather than in special test procedures, knowledge of these distinctions becomes directly manipulatable by various LIFER procedures. In particular, LIFER has more information from which to generate user-meaningful error messages, and upon which to base elliptical analysis and paraphrase generalization. On the other hand, use of the *ERROR* feature leads to smaller grammars.

2. Left Recursion

There are no restrictions on the types of patterns that may be used in defining subgrammars. In particular, left recursion is permitted, e.g., <A> => (<A> A). To allow left recursion, the LIFER parser optionally traps entries into subgrammars that would cause the level of recursion to exceed a given depth. This depth, which is the value of global variable LIFER.MAXDEPTH, is initially set to 6, but may be changed to meet the needs of a particular application language. The lower the number, the more efficient the parser's operation. In particular, if left recursion is not to be used, LIFER.MAXDEPTH should be set to zero. Only left recursion is affected by LIFER.MAXDEPTH. Other forms of recursion may extend to arbitrary depths (being limited "naturally" by the number of words in the input string).

In general, left recursion leads to inefficient processing and system builders are advised to avoid it. If the system builder finds LIFER accepts short (shallow) inputs but rejects longer (deeper) inputs, it is likely that the value of LIFER.MAXDEPTH is too low.

3. Redifining and Editing

If PD is called more than once with the same pattern for the same meta symbol, a message will be printed asking whether the response expression for the pattern should be redefined.

The user may type either YES or NIL. (If the user doesn't answer this question within 30 seconds, the system answers YES.) Response expressions will be redefined without the printing of error messages if the flag REDEFINE.PATTERNS is non-NIL.

To edit the total collection of patterns and response expressions associated with a meta symbol, use the function EDIT.GR, described in Section IX.

4. Compiling Response Expressions

To increase the run time efficiency of the LIFER system, all but the most trivial response expressions are automatically converted into calls to functions of no arguments. This conversion, which is optional, allows the response expressions to be compiled. (See discussion of SAVE.GRAMMAR below.) The rules for expression conversion are the following. Atoms are never converted. Single member lists (i.e., calls to functions of no arguments) are not converted. Any S-expression whose CAR is QUOTE is not converted. Lists of multiple elements of the form

```
(FUN ARG1 ARG2 ... ARGn)
```

are replaced by

```
(new.function)
```

where new.function is a newly created function defined as

```
(LAMBDA NIL (FUN ARG1 ARG2 ... ARGn)
```

A list of names of functions defined in this manner is saved as the value of the global variable LIFER.FUNCTIONS. If the same response expression is used in the definitions of multiple patterns, all occurrences will use the same new function. Converson of response expressions to functions may be turned off by setting FUN.FLAG to NIL.

5. Efficiency Advice

A useful method for increasing parse time efficiency is to delay costly computations until a top-level pattern has been accepted. Stated in the negative, costly computations in response expressions associated with subgrammars should be avoided, because the parser is likely to accept temporarily some subgrammars that will later be discarded. One method for delaying computations is to have subgrammers return expressions for computing values rather than the values themselves. Such expressions need be evaluated only after a top-level pattern has been accepted.

IV. USING THE LIFER PARSER

The LIFER parser may be invoked as soon as even one top-level sentence pattern has been specified. There are multiple methods for sending information to the parser. The first method is convenient for intermixing calls to the parser with ordinary top-level requests to INTERLISP. When the INTERLISP prompt character is typed, the user may type a semicolon, the sentence to be processed by the parser, and a carrage return. For example, if the INTERLISP prompt character is "-" and the input sentence is "How old is John", then the interaction with LIFER would look like this:

```
-; HOW OLD IS JOHN (carriage return)
PARSED!
15 YEARS OLD
```

The semicolon need not be separated from the sentence by a blank. The semicolon is only one of a number of initial control characters that may be used to invoke the LIFER parser and affect parsing behavior, but it enables the most general set of features.

If all inputs are (for a time at least) to be given to the parser and not interpreted by INTERLISP, then the user may set the flag LIFER.INPUT to the character ";". This may be accomplished by the function call

```
(LIFER.INPUT)
```

After this change, EVERY line typed into the system will be sent to the parser. Such inputs may omit the initial semicolon (but need not). To return to normal INTERLISP processing, the input sentence

```
RESTORE
```

may be used. This sentence is predefined at the top-level of the application language when LIFER is initialized.

The parser may be invoked directly by the call

```
(PARSE input.list)
```

where input.list is a list of words composing a (possible) sentence. For example, to parse the sentence

HOW OLD IS JOHN

function PARSE may be called directly as in

```
(PARSE '(HOW OLD IS JOHN))
```

The function SUBPARSE provides an entry into the parsing procedures that is useful for the defining of certain predicates. (See discussion of predicates above.) Calls to this function are of the form

```
(SUBPARSE meta.symbol input.list)
```

where input.list is to be parsed in accordance with the grammar named by meta.symbol. If the parse fails, SUBPARSE returns the atom *ERROR*. If successful, the value of the matching pattern's response expression (which may be NIL) is returned. SUBPARSE suppresses the printing of all parser messages, including error messages. If the user has defined a grammar for chemical formulas, then a call to SUBPARSE such as

```
(SUBPARSE '<CHEMICAL> '(H 2 O))
```

might be appropriate.

The parser operates in a top down, left to right mode. As an input is processed, the teletype print head or the display cursor will follow the progress of the parsing by positioning itself beneath the word currently being interpreted. (When the time-sharing system is heavily loaded, this feedback assures the user that the input is being processed.) When parsing is complete, LIFER types the message PARSED! before evaluating the top-level response function. This message lets the user know that the input has been "understood" and that response computations are underway.

V. THE ELLIPSIS FEATURE

Because it becomes tiresome to type complete input sentences when several inputs involving the same input pattern are to be processed in sequence, LIFER provides an ellipsis (incomplete sentence) processing facility to permit abridged inputs that match only portions of existing patterns. For example, suppose the pattern

```
(HOW <ADJ> IS <PERSON>)
```

is contained in the system and that the user wishes answers to the questions

```
HOW OLD IS JOHN
HOW TALL IS JOHN
HOW TALL IS MARY
```

Rather than type out each of these queries in full, it takes less effort and is more natural to type just

```
HOW OLD IS JOHN
HOW TALL
MARY
```

The first query in the latter sequence matches the pattern cited above and is processed in the normal way. However, HOW TALL does not match the pattern. If in fact there is no sentence-level pattern in the system that matches HOW TALL, the parser turns processing over to a special ellipsis routine. This routine remembers the pattern (or patterns) used in processing the last acceptable input and attempts to match the current input against the various contiguous pattern fragments that may be extracted from the old pattern (and the patterns of subphrases that were recognized by subgrammars). If the system has just processed HOW OLD IS JOHN, then the old pattern is (HOW <ADJ> IS <PERSON>). A fragment of this old pattern is

(HOW <ADJ>),

which matches HOW TALL. By supplementing the current input with information extracted from the last, the ellipsis routine expands HOW TALL into HOW TALL IS JOHN.

Similarly, with the system remembering the (expanded) input HOW TALL IS JOHN, the abridged input MARY is matched against the pattern fragment

(<PERSON>)

and expanded into HOW TALL IS MARY.

The user who watches the terminal display during the processing of an elliptical input may see the cursor or print head move as the parser tries to match the input against a complete pattern at the sentence level. Once the parser has given up on a sentence-level match, the message TRYING ELLIPSIS: will be printed. If the ellipsis facility succeeds in finding a partial match, the expanded interpretation of the input is printed so that the user may know whether or not the incomplete input was expanded as intended.

For many application languages, the time spent in determining that an input does not fit a sentence-level pattern is relatively small. However, the user may skip this process and try ellipsis directly by typing a comma at the beginning of the input line in place of the semicolon discussed above.

It is often possible for the ellipsis routines to match an input against multiple fragments of the previous input. However, the system will make that substitution that is leftmost in the sentence sequence and at the least possible depth of recursion. Consider, for example, the pattern

(IS <PERSON1> <CADJ> THAN <PERSON2>)

which was discussed above. In the context of input

IS JOHN TALLER THAN SUE

the input

SAM

will be expanded into IS SAM TALLER THAN SUE. Although SAM is capable of matching both <PERSON1> and <PERSON2> in the old pattern, the ellipsis facility makes the substitution for <PERSON1> because it is the leftmost of the possibilities. It should be noted that language users are more likely to ask "IS SAM" or "THAN SAM" than "SAM," because humans implicitly realize the ambiguity of "SAM" and, through years of natural language training, are in the habit of providing clues for disambiguation. LIFER will use such clues when they are furnished.

Using the pattern

(IN WHICH DEPARTMENT DO <PEOPLE> WORK)

the input

IN WHICH DEPARTMENT DO JOHN AND TOM AND MARY WORK

will cause the meta symbol <PEOPLE> to be expanded on multiple levels. Thus the input

SUE AND GENIE

conceivably could be expanded into such interpretations as

* IN WHICH DEPARTMENT DO JOHN AND TOM AND SUE AND GENIE WORK.

By forcing substitution at the least depth of recursion, the system will, in fact, interpret SUE AND GENIE as

IN WHICH DEPARTMENT DO SUE AND GENIE WORK,

which in most cases seems to be the substitution preferred by humans.

The system allows elliptical inputs to begin in the middle of one subgrammar and end in the middle of another. To see this, suppose the pattern

(NP VP)

appears at the top-level and that NP may be expanded into

(WHAT <CLASS.NOUN> FROM <PLACE>)

and VP into (<VERB> <OBJECT>). Given the input

WHAT PEOPLE FROM FACTORY F MAKE SHOES

the subsequent input STORE Q SELL will expand into WHAT PEOPLE FROM STORE Q SELL SHOES. A couple of additional possibilities are: MACHINES goes to WHAT MACHINES IN FACTORY F MAKE SHOES; SEW goes to WHAT PEOPLE FROM FACTORY F SEW SHOES.

VI. SPELLING CORRECTION AND PARSER ERROR MESSAGES

As the parser works its way through an input, it remembers those points at which it fails and is forced to back up. If the parser fails to find any path through the patterns of the application language that will match the input, then this history of failpoints is used to aid recovery processes.

For complete inputs (or inputs that are at first assumed to be complete), the first type of failure processing attempted by the parser is spelling correction. When LIFER believes a word to be misspelled, it will type the "correct" spelling immediately below the misspelled word and resume the parsing process.

For inputs preceded by the semicolon control character, the rejection of an attempted analysis for a complete sentence or question will automatically cause elliptical processing to be invoked.

If all attempts at parsing and error recovery are unsuccessful, then the history of failpoints is used to give the user some guidance concerning where his input failed to meet the language definition. Because it is normal for the parser to explore a number of false paths in interpreting a sentence, a heuristic is applied to determine which failpoint to report to the user. This failpoint is the rightmost (and, in case of tie, shallowest) failpoint in the failpoint history.

The parser recognizes three major categories of errors. In the first, the parser is able to account for the entire input, but needs more input on the right to complete a pattern. That is, the parser can find a pattern that swallows up the input, but the input is exhausted before some of the symbols at the end of the pattern are matched. For this error, the system prints a list or those words and meta symbols that may be used to extend the input.

In the second type of error, the parser finds a word in the input that it cannot interpret in the context of those words appearing to the word's left. The system indicates to the user what words or meta symbols may be recognized in that context. The user may find which terminal words match a meta symbol by using the function SYMBOL.INFO discussed below.

In the third type of error, the parser finds an acceptable syntactic analysis of the input, but response expressions associated with some syntactic unit have returned *ERROR*, causing the analysis to be rejected on semantic grounds. Because LIFER is syntactically oriented, it can provide no real help for users confronted with this type of error.

VII. INITIAL CONTROL CHARACTERS

After LIFER has been loaded into INTERLISP, the first character of each line typed by the user is examined to determine how the input is to be treated by the system. If the first character is taken from the set of symbols {!;:,@+*}, then processing follows the corresponding rule cited below. If the initial character is not a member of this set, then the value of the global variable LIFER.INPUT is used as the control character. If this value is not in the set, character "!" is used. Normally, an interface builder should set LIFER.INPUT to one of these characters so that the actual end user need not be concerned with input control.

! -- Input line is given to LISP for normal processing.

; -- Attempt to parse input as complete sentence.
If this fails, try to do spelling correction.
If this fails, try elliptical processing.

: -- Attempt to parse input as complete sentence.
If this fails, try spelling correction, but no ellipsis.

@ -- Attempt to parse input as complete sentence.
Don't try spelling correction or ellipsis.

, -- Attempt to parse as an elliptical input.

+ -- resume last input at the point which caused the error.

* -- Treat line as a comment.

The control character feature of LIFER is implemented via a

LISPXUSERFN. Hence, system builders wishing to use their own LISPXUSERFN should ADVISE the one supplied by LIFER.

VIII. THE PARAPHRASE FEATURE

A. The Function PARAPHRASE

The function PARAPHRASE allows naive users to expand the language definition without knowing the underlying grammar or the nature of the language specification routines. PARAPHRASE takes three arguments: NEW-PHRASING, OLD-PHRASING and META-SYMBOL. If META-SYMBOL is NIL, then OLD-PHRASING should be a legal sentence-level input in the application language. If META-SYMBOL is non-NIL, then OLD-PHRASING should be a legal expansion of META-SYMBOL. In either case, hereafter the parsing of NEW-PHRASING is to produce the same effect as the parsing of OLD-PHRASING.

As an example of the use of PARAPHRASE, assume that

```
    (WHAT <AUXB> THE <ATTRIBUTE-SET> OF <PEOPLE>)
```

is a pattern at the sentence level and that <ATTRIBUTE-SET> and <PEOPLE> may be expanded as follows:

```
     <ATTRIBUTE-SET> => (<ATTRIBUTE>)
                     => (<ATTRIBUTE> AND <ATTRIBUTE-SET>)
            <PEOPLE> => (<PERSON>)
                     => (<PERSON> AND <PEOPLE>)
```

Assume also that <AUXB> includes members of the set {IS ARE}, that <ATTRIBUTE> includes members of {AGE HEIGHT WEIGHT}, and that <PERSON> includes members of {JOHN TOM SUE}. Then the system will accept such inputs as

```
WHAT ARE THE AGE AND WEIGHT OF TOM
```

To expand the language specification without having to mention such things as meta symbols or patterns, PARAPHRASE may be called as follows:

```
PARAPHRASE((PRINT AGE FOR TOM)
           (WHAT IS THE AGE OF TOM))
```

PARAPHRASE creates new structures that will cause the LIFER parser to recognize

```
PRINT AGE FOR TOM
```

as a paraphrase of

```
WHAT IS THE AGE OF TOM
```

In particular, PARAPHRASE creates the new top-level pattern

```
(PRINT <ATTRIBUTE-SET> FOR <PEOPLE>)
```

and an appropriate response expression for carrying out the commands that match the new pattern. The naive user need know nothing about the new pattern and its associated response expression.

Note that the new pattern generated by PARAPHRASE will match many more inputs than just NEW-PHRASING. For the current example, such inputs as

```
PRINT THE AGE AND HEIGHT FOR SUE AND JOHN
```

will be matched and produce appropriate responses.

Sometimes the function PARAPHRASE makes changes in subgrammars, even when it is called with respect to the sentence-level grammar. For example, consider

```
   PARAPHRASE((WHAT ARE THE AGE AND HEIGHT OF THE BOYS)
              (WHAT ARE THE AGE AND HEIGHT OF JOHN AND TOM))
```

PARAPHRASE recognizes that the difference in these inputs can be accounted for solely in terms of a change in <PEOPLE>. Therefore, it asks

```
          MAY LIFER ASSUME THAT "THE BOYS" MAY ALWAYS BE USED
          IN PLACE OF "JOHN AND TOM"?
```

If the user answers YES, PARAPHRASE extends the definition of <PEOPLE> to include the pattern

```
                           (THE BOYS),
```

and indicates this extension to the (more sophisticated) user by typing

```
                    <PEOPLE>  =>  (THE BOYS).
```

The phrase THE BOYS will subsequently match <PEOPLE> in any pattern in which <PEOPLE> appears. In particular, the input

```
                    PRINT HEIGHT FOR THE BOYS
```

will now be parsed and interpreted as

```
               WHAT IS THE HEIGHT OF JOHN AND TOM
```

Suppose the user types NO to the system-generated query above. Then PARAPHRASE assumes only that THE BOYS means JOHN AND TOM in the context of queries following the pattern

```
         (WHAT <AUXB> THE <ATTRIBUTE-SET> OF THE BOYS).
```

Therefore, PARAPHRASE extends the sentence-level grammar to include this new pattern. The response expression that PARAPHRASE creates to answer queries following this pattern will find the attributes of the <ATTRIBUTE-SET> for exactly the constants JOHN and TOM.

B. Accessing PARAPHRASE Through Natural Language

The reason for having a PARAPHRASE function is to make it easy for computer-naive users to extend and personalize an application language without having to know anything about formal language specification procedures. Therefore, if users are forced to create their own explicit calls to this function, its utility will be largely lost. Thus, it is strongly recomended that interface builders provide a natural language link to PARAPHRASE. For example, the calls

```
PD((LET <S1> BE A PARAPHRASE OF <S2>)
   (PARAPHRASE <S1> <S2>))
MAKE.PREDICATE(<S1> LISTP)
MAKE.PREDICATE(<S2> LISTP)
```

will create the internal structures necessary to allow users to simply say things such as

```
LET (PRINT AGE FOR JOHN) BE A PARAPHRASE OF (WHAT IS THE
AGE OF JOHN)
```

If a predicate STRING.TO.LIST that recognizes strings and converts them to lists is used in place of LISTP, then the more natural input

LET "PRINT AGE FOR JOHN" BE A PARAPHRASE OF "WHAT IS THE
AGE OF JOHN"

may be used.

IX. AUXILIARY FEATURES

The central functions are supported by a number of auxiliary routines.

A. GRAMMAR.ANALYSIS

One of these is the function GRAMMAR.ANALYSIS, which causes a complete and easily readable discription of the current application language to be written on a file. Arguments to GRAMMAR.ANALYSIS are FILE.NAME and WIDTH, where WIDTH is the maximum line length to be used in writing the file. FILE.NAME defaults to the name GRAMMAR.ANALYSIS and WIDTH defaults to 72.

B. PRINT.GRAMMAR

The function PRINT.GRAMMAR may be used to print the production patterns associated with the top-level of the application language or any of its subgrammars. The function takes two arguments: SYMBOL and FLAG. SYMBOL may be a meta symbol whose associated patterns are to be printed. It may also be the atom LIFER.TOP.GRAMMAR or NIL, in which case the top-level patterns are printed. If FLAG is non-NIL, the response expression (or its functional conversion, see Section III.F.4) associated with each pattern is also printed. The grammars are printed as sequences of production rules of the form

Meta-Symbol => pattern

C. SYMBOL.INFO

The function SYMBOL.INFO takes a meta symbol as its argument and prints all the ways in which that meta symbol may be matched. This includes sets, predicates and subpatterns. Because various error messages make reference to meta symbols, parser users may wish to obtain information about them. For those users who do not know INTERLISP, the system builder may wish to provide natural language access to the SYMBOL.INFO function. To do this, a pattern such as

(HOW IS <SYMBOL> USED)

may be defined with response expression

(SYMBOL.INFO <SYMBOL>).

LIFER remembers which meta symbols have been defined in terms of sets, predicates, and patterns (or combinations of these) by maintaining lists, which are the values of global variables LIFER.SETS, LIFER.PREDICATES, and LIFER.GRAMMARS. Thus, the symbol <SYMBOL> may be defined by using the predicate

```
    (LAMBDA (W) (AND (OR (FMEMB W LIFER.SETS)
                         (FMEMB W LIFER.PREDICATES)
                         (FMEMB W LIFER.GRAMMARS))
                     W))
```

D. User-Supplied Functions

Provision has been made in LIFER for two functions to be defined by the interface builder. (Standard functions are provided, which act as no-ops.) These are USER.NEW.WORD and USER.PREPROCESSOR.

When new symbols and grammar patterns are being defined, the function USER.NEW.WORD is called whenever the system thinks it might be seeing a word for the first time that is to become a part of the language. These calls make it possible for the user to collect a word list or do other processing as desired. USER.NEW.WORD has two arguments, the new word and a flag. If the flag is NIL, the word is being defined as the possible value of a meta symbol through MAKE.SET. If T, the word is being defined as a direct member of some pattern. USER.NEW.WORD may be called more than once with the same word.

The second user function, USER.PREPROCESSOR, is a function of one argument that is called by the parser before parsing actually begins. Its argument is the list of words that have been given to the parser as an input. USER.PREPROCESSOR may convert this list into another list that will actually undergo the parsing process. This feature of LIFER gives the user the opportunity to implement lexical strippers (for example, to convert plurals to roots plus suffix as in BOYS => BOY -S). For processing highly inflected languages, such as German, this feature is much more crucial than for English.

E. SAVE.GRAMMAR

The function SAVE.GRAMMAR has been provided to save language definitions on files. Like MAKEFILE, of INTERLISP, SAVE.GRAMMAR takes a file name as its argument. An atom fileCOMS is created (or found) following the usual INTERLISP conventions. If fileCOMS is a new atom, a value is assigned to it that will cause the language definition to be saved. If fileCOMS is a commands list that the user has set up, provisions are added to the list that will cause the language definition to be saved. If fileCOMS was produced by either of the above methods, subsequent calls to SAVE.GRAMMAR need make no alterations in fileCOMS. If the user has defined user functions and has them on source file USER.LSP, it may be convenient to save the language definition on a new version of this same file by calling SAVE.GRAMMAR with USER.LSP as the argument.

To load a language definition written by SAVE.GRAMMAR, the LIFER system should be loaded first and then the file on which the language definition was saved should be loaded (using the ordinary INTERLISP LOAD function).

The file made by SAVE.GRAMMAR may be compiled (use TCOMPL) to improve execution time for response expressions. (However, to avoid multiple function definitions, if the same response expression is used with multiple patterns, all such patterns should be specified before compiling.)

F. EDIT.GR

The function EDIT.GR takes a SYMBOL as its argument and converts the internal transition network defining the subgrammar named by SYMBOL (use NIL for the sentence level grammar) into a list of pairs of the form (pattern response). This list is then given to the INTERLISP editor. The interface builder may change pairs, add new pairs or delete pairs, using the full power of the editor. When the interface builder types OK, the edited list is reconverted into internal networks. This function is extremely useful in constructing more complex language specifications and its use is strongly encouraged.

G. Other Functions

Other LIFER functions include:

EDIT.RESPONSE.EXPRESSION(PATTERN SYMBOL).
Nickname is EDIT.RE. For the grammar called SYMBOL, the response expression associated with PATTERN is retrieved. The user may then edit this expression.

EXPUNGE.ELEMENTS(SYMBOL LIST). Nickname is EE. The items on LIST are expunged from the set associated with SYMBOL. Inverse of MAKE.SET.

EXPUNGE.PATTERN(PATTERN SYMBOL). Nickname is EP. PATTERN is no longer to be an expansion of SYMBOL.

GET.RESPONSE.EXPRESSION(PATTERN SYMBOL). Nickname is GRE. For subgrammar SYMBOL, finds the response expression associated with PATTERN.

MPQ(LIST). An NLAMBDA function, MPQ applies MAKE.PREDICATE to each of the items on LIST.

MSQ(LIST). An NLAMBDA function, MSQ applies MAKE.SET to each of the items on LIST.

PDQ(LIST). Each item on LIST is of the form (SYMBOL plist), where plist is a list of pairs of the form (PATTERN RESPONSE). For each pair, a call to PATTERN.DEFINE is made of the form

PD(PATTERN RESPONSE SYMBOL).

PDQ is an NLAMBDA.

PATTERN.REFERENCES(SYMBOL). Prints production rules indicating the patterns in which SYMBOL is used.

REDEFINE.PATTERN(PATTERN RESPONSE SYMBOL). Like PATTERN.DEFINE, but does not print error messages when a pattern is given a new response expression.

SYNTAX(BINDINGS.FLG). Prints the syntax tree of the last sentence parsed. If BINDINGS.FLG is T, the value of each nonterminal is displayed.

X. IMPLEMENTING PRONOUNS

Pronouns (and, more generally, determined noun phrases) furnish many difficult problems for language definition designers. LIFER supplies no simple solution to these problems. However, a few observations may be of help .

First, there are many trivial uses of pronouns in which it is not necessary to invoke heavy machinery to resolve the reference. Examples include

```
WHAT TIME IS IT
IS IT RAINING OUTSIDE
```

and instances in which the pronoun refers to an earlier word in a pattern

```
    (DOES THE <THING> HAVE ALL ITS <PARTS>)
            as in DOES THE CHAIR HAVE ALL ITS LEGS
                  DOES THE RADIO HAVE ALL ITS TUBES
    (DID <PERSON-POSSESSIVE> <RELATION> <TRANS-VERB> HIM)
            as in DID JOHN'S BOSS FIRE HIM
                  DID SAM'S DOG BITE HIM
```

Very often pronouns are used in natural language to refer to things mentioned in the "last sentence." Thus, in the sequence

```
                WHAT IS THE LENGTH OF THE SEAWOLF
                WHAT IS ITS SPEED
```

the pronoun ITS refers to THE SEAWOLF. Suppose the first sentence above is interpreted by means of the pattern

```
               (WHAT IS THE <ATTRIBUTE> OF <SHIP>)
```

and the second sentence by

```
             (WHAT IS <SHIP-POSSESSIVE> <ATTRIBUTE>)
```

The primary method for matching <SHIP-POSSESSIVE> might be through a pattern such as

```
                        (THE <SHIP> -'S)
```

where -'S is the possessive forming suffix which is stripped off by a preprocessor as described above. (Alternatively, a set of possessive nouns naming ships could be defined and the stripper not used.) In addition to this pattern, <SHIP-POSSESSIVE> may also be defined to match ITS if a <SHIP> was used in the last input. This will allow WHAT IS ITS SPEED to be interpreted as

WHAT IS THE SEAWOLF'S SPEED. To define <SHIP-POSSESSIVE> in this way, use

```
    MAKE.PREDICATE(<SHIP-POSSESSIVE>
                   SHIP.ITS)
```

where SHIP.ITS is a predicate defined by

```
    (LAMBDA (WORD)(AND (EQ WORD 'ITS)
                       (LIFER.BINDING '<SHIP>)))
```

LIFER.BINDING is a special LIFER function that determines whether the meta symbol given as an argument had a binding in the interpretation of the last input. If so, the binding is returned. (If there were multiple occurences of the symbol in the last input, the leftmost-topmost instance is returned).

Because pronouns sometimes are used to refer to the RESPONSE to the last input, the interface builder may wish to assign meta symbols to the responses. For example, consider the sequence

```
        WHAT SHIP IS COMMANDED BY CAPTAIN SMITH
        WHAT IS ITS SPEED
```

Here ITS refers to the ship which Captain Smith commands. To use the ITS recognition predicate above, meta symbol <SHIP> must be bound to the answer returned by the first input. One way to do this is to make

```
                (<SHIP>)
```

a top-level pattern and to make

```
        (WHAT SHIP IS COMMANDED BY <PERSON>)
```

a pattern defining <SHIP>.

To allow elliptical inputs which might be matched by <SHIP>, the pattern (<SHIP>) must not be used at the top-level. Hence, to allow ellipsis, a new symbol <SHIP*> may be defined for use in top-level pattern (<SHIP*>) and SHIP.ITS may be redefined as

```
        (LAMBDA (WORD) (AND (EQ WORD 'ITS)
                            (OR (BINDING '<SHIP>)
                                (BINDING '<SHIP*>) )))
```

Another technique, which works nicely for some classes of anaphoric references, involves the use of global variables (sometimes called "registers"). For example, suppose that each response expressions associated with a pattern defining the meta symbol <SHIP> is so constructed that it will set the global variable LATEST-SHIP to the value it returns as the binding of <SHIP>. To be concrete,

```
          PD(<SHIP>
             (<SHIP.NAME>)
             (SETQ LATEST-SHIP <SHIP.NAME>))
```

causes <SHIP> to match the pattern (<SHIP.NAME>). The response expression that computes the value of <SHIP> will return the value of <SHIP.NAME>, but, as a side effect, it will also set the global variable LATEST-SHIP to this same value. Later, when phrases such as THE SHIP or THAT SHIP are used to refer to the last ship mentioned, the global variable LATEST-SHIP may be used to recall the last ship. For example, if <DET-DEF> is the set of definite determiners (i.e., THAT, THE, etc.), then

```
PD(<SHIP>
   (<DET-DEF> SHIP)
   LATEST-SHIP)
```

will define structures that allow <SHIP> to match THE SHIP and to take as its value the value of the LATEST-SHIP. Note carefully that LATEST-SHIP is always ready with the value of the last <SHIP> mentioned, but (LIFER.BINDING '<SHIP>) is of no value if the immediately preceding input did not use a <SHIP>.

XI. CURRENT SYSTEM IMPLEMENTATION

LIFER is implemented in PDP-10 INTERLISP, with the basic system requiring an additional 14K words above the 150K used by INTERLISP. An extensive language definition for communicating with a large data base (100 fields on 14 files with hundreds of records) requires an additional 33K, including some data base access routines. Such sentences as

WHAT IS THE LENGTH OF THE FASTEST AMERICAN SUBMARINE

parse in less than .2 seconds of CPU time, using block-compiled INTERLISP on the DEC KL-10. This is much faster than the sentences are usually spoken or typed.

Appendix A
FUNCTION LIST

EDIT.GR[SYMBOL]. Gives control to the INTERLISP editor to edit the productions expanding SYMBOL.

EDIT.RESPONSE.EXPRESSION[PATTERN SYMBOL]. Gives control to the INTERLISP editor to edit the response expression associated with PATTERN in the expansion of SYMBOL.

EXPUNGE.ELEMENTS[SYMBOL LIST]. The items on LIST are removed from the set of words which may match SYMBOL.

EXPUNGE.PATTERN[PATTERN SYMBOL]. PATTERN is no longer to be an expansion of SYMBOL.

GRAMMAR.ANALYSIS[FILE WIDTH]. Writes an analysis of the current language definition on FILE. WIDTH specifies line length of printing.

GET.RESPONSE.EXPRESSION[PATTERN SYMBOL]. Gets the response expression associated with the expansion of SYMBOL as PATTERN.

LIFER.BINDING[SYMBOL]. Returns the value of the leftmost, topmost occurrence of SYMBOL in the syntactic analysis of the last sentence. If SYMBOL did not occur, returns NIL.

LIFER.INPUT[]. Equivalent to (SETQ LIFER.INPUT ';)

MAKE.PREDICATE[SYMBOL PREDICATE]. SYMBOL is to match any S-expression satisfying PREDICATE.

MAKE.SET[SYMBOL SET.SPECIFICATION]. SYMBOL is to match words and phrases as indicated by SET.SPECIFICATION. See Section III.C.

MPQ[LIST]. An NLAMBDA. Each item of LIST is an argument list for a call to MAKE.SET.

MSQ[LIST]. An NLAMBDA. Each item of LIST is an argument list for a call to MAKE.SET.

PARAPHRASE[NEW.VERSION OLD.VERSION SYMBOL]. If OLD.VERSION matched SYMBOL (which, if NIL, may be the top-level syntax), then NEW.VERSION will henceforth match SYMBOL also, with the same interpretation.

PARSE[INPUT.LIST ELLIPSIS.FLG ELLIPSIS.ONLY.FLG SPELLING.FLG RESTART.FLG]. INPUT.LIST is the input to be parsed. Other arguments control the parsing. If all flags are NIL, PARSE attempts to parse INPUT.LIST as a complete sentence, without spelling correction. SPELLING.FLG turns on the spelling corrector (for nonelliptical inputs only). If ELLIPSIS.FLG is T, an input that fails to parse at the top level will be given to the ellipsis routines. If ELLIPSTS.ONLY.FLG is also T, the INPUT.LIST goes directly to ellipsis routines. If RESTART.FLG is T and an error occurred in the previous call to PARSE, parsing is resumed at the previous fail point, using the current INPUT.LIST as the new tail of the input sentence.

PATTERN.DEFINE[A1 A2 A3]. Either A1 is a SYMBOL, A2 is a PATTERN, and A3 is a RESPONSE.EXPRESSION, or A1 is a PATTERN, A2 is a RESPONSE.EXPRESSION, and A3 is a SYMBOL. PATTERN is to be an expansion of SYMBOL, whose value is to be computed by RESPONSE.EXPRESSION. For sentence-level expansions, use NIL for SYMBOL.

PATTERN.REFERENCES[SYMBOL]. Prints all productions in which SYMBOL appears in an expansion pattern.

PDQ[LIST]. An NLAMBDA. LIST is a list of items of the form (SYMBOL PRLIST), where PRLIST is a list of PATTERN-RESPONSE.EXPRESSION pairs. For each item on LIST, for each pair on PRLIST, a call is made to PD of the form PD[SYMBOL PATTERN RESPONSE.EXPRESSION]..

PRINT.GRAMMAR[SYMBOL RESPONSE.FLG]. Prints the productions expanding SYMBOL. If RESPONSE.FLG is T, the associated response expressions are printed also.

REDEFINE.PATTERN[A1 A2 A3]. Like PATTERN.DEFINE, but doesn't give error messages when a pattern is redefined with a new response.

SAVE.GRAMMAR[FILE]. Prints out the current language definition on FILE. The language definition may subsequently be reloaded using LOAD.

SUBPARSE[SYMBOL INPUT.LIST]. Parses INPUT.LIST using the productions expanding SYMBOL.

SYNTAX[BINDINGS.FLG]. Prints the parse tree of the last input parsed. If BINDINGS.FLG is T, the values of nonterminals are displayed.

USER.NEW.WORD[WORD FLAG]. A function to be supplied by the user. (The standard version is a no-op.) Called whenever a new WORD is defined through MAKE.SET or PATTERN.DEFINE. FLAG is T if word is encountered during a call to PATTERN.DEFINE.

USER.PREPROCESSOR[INPUT.LIST]. A function to be supplied by the user. (The standard version simply returns INPUT.LIST.) PARSE calls USER.PREPROCESSOR with its own INPUT.LIST argument. Parsing is actually performed on the output from USER.PREPROCESSOR.

Appendix B
FUNCTION ABBREVIATIONS

ABBREVIATION	FULL FUNCTION NAME
EDIT.RE	EDIT.RESPONSE.EXPRESSION
EE	EXPUNGE.ELEMENTS
EP	EXPUNGE.PATTERN
GRE	GET.RESPONSE.EXPRESSION
LB	LIFER.BINDING
MP	MAKE.PREDICATE
MS	MAKE.SET
PD	PATTERN.DEFINE
PG	PRINT.GRAMMAR
PR	PATTERN.REFERENCES
RP	REDEFINE.PATTERN

Appendix C
GLOBAL PARAMETERS

I. Parameters That May Be Set

FUN.FLG
If T, response expressions are converted to functions of no arguments, which may be compiled.

LIFER.MAXDEPTH
Maximum depth of left recursion. Originally set to 6.

LIFER.PROMPTCHAR.LENGTH
Length of the INTERLISP prompt. Originally set to 2. Should be set to the column number where users begin typing their inputs to the parser.

II. Parameters That May Be Accessed

LIFER.FUNCTIONS
List of functions created by LIFER to replace response expressions.

LIFER.GRAMMER
List of meta symbols that may be expanded by patterns.

LIFER.PREDICATES
List of meta symbols that may be matched by satisfying predicates.

LIFER.SETS
List of meta symbols that may match members of explicit sets.

LIFER.TIME
CPU milliseconds required to parse the last input.

OLD.ANSWER
Value returned by top-level response expression for last parse.

OLD.INPUT
Input to the last parse.

REFERENCES

1. Hendrix, G. G. (1977), "Human Engineering for Applied Natural Language Processing," Technical Note 139, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California.

2. Sacerdoti, E. D. (1977), "Language Access to Distributed Data with Error Recovery," Technical Note 140, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California.

3. Teitelman, W. (1975), Interlisp Reference Manual, Xerox Palo Alto Research Center, Palo Alto, Calfornia.

4. Woods, W.A. (1970), "Transition Network Grammars for Natural Language Analysis," CACM 13 (10), 591-606.