MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 185 April, 1979

Building English Explanations from Function Descriptions

Bruce Roberts

ABSTRACT. An explanatory component is an important ingredient in any complex AI system. A simple generative scheme to build descriptive phrases directly from Lisp function calls can produce respectable explanations if explanation generators capitalize on the functional decomposition reflected in Lisp programs.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-77-C-0389.

Knowledge-based systems must explain themselves.

Intelligent systems become increasingly difficult to understand as they acquire more and more facts and grow steadily more complex in their operation. The need to understand a system is obvious for the system designer, but this need is equally valid for the users if they are to place confidence in the system's judgments. Systems that are able to explain themselves must be built to satisfy this need for intelligibility.

The atomic names and S-expressions that compose a system's internal structure inadequately describe the system's knowledge. Instead, concepts represented in a system can have associated descriptions understood by the system's users. However, a static description is neither as useful nor precise as one produced to explain a particular instance of a concept. The descriptions must be sensitive to the environment in which a concept appears.

A portion of a sample dialogue with the SHIPMAINT system illustrates the application of a context-sensitive description mechanism. In this dialogue, a supervisor is interacting with SHIPMAINT to schedule maintenance on a ventilating fan of a ship. SHIPMAINT has access to knowledge about what facilities are required to repair fans and a preference for where the repair is to take place. This portion of the dialogue is generated by a procedure designed to elicit the supervisor's specification for the place of the repair:

> "Repairing fans requires access to an electrical shop. Two are available: the Electrical Shop on Deck B and the Electrical Shop on Deck C. One prefers that *the deck of the place is the same as the deck of the location of the object.*"

The italicized phrase is a preference represented in SHIPMAINT by a LISP predicate. The preference predicate is defined to compute the validity of the relation and is used by SHIPMAINT to do its own scheduling. However, an English phrase is understood more easily than the predicate in any explanation of SHIPMAINT's behavior, as in this example of suggesting constraints on the repair specifications to the supervisor.

Knowledge in SHIPMAINT is represented as a network of frames. The preference predicate from the example appears in a frame that describes the general task of repairing objects on a ship. The Task frame represents a commitment of resources -- people, objects and places -- for a particular time interval:

```
(TASK
   (TIME ...)
   (OBJECT ...)
   (PARTICIPANT ...)
   (PLACE ($PREFER
            (EQUAL (DECK :VALUE)
                   (DECK (FGET-VALUE (FGET-VALUE :FRAME 'OBJECT)
                                     'LOCATION)))))))
```

In SHIPMAINT, the predicate corresponding to the italicized phrase in the example appears in the Prefer facet for the Place slot of the Task frame. The predicate contains an ordinary LISP function (EQUAL), a user-defined function (DECK), variable names (:VALUE and :FRAME) and a frame retrieval function (FGET-VALUE). :VALUE is a symbol bound by convention to the Place slot's value when the preference is evaluated. :FRAME is similarly bound at evaluation time to provide access to the current frame environment. FGET-VALUE returns the value of a frame's slot.

Each function and variable describes itself.

The problem of getting SHIPMAINT to explain preferences is just one example of a need to turn internal S-expressions into external prose. The designer of any system assigns an interpretation to each of a program's functions and variables. These explicit interpretations appear in the documentation of the program for the benefit of the program's readers, but do not benefit the program's users. How can functions and variables be made self-documenting wherever they appear?

Good programming style dictates that a variable retain its unique meaning wherever it is used; therefore it should be possible to associate with a variable name a descriptive phrase to be substituted for the name when talking about its use. The property list of an atom used as a variable can store the descriptive phrase on the SAY-VALUE property. Actually, so as not to be limited exclusively to canned phrases, the SAY-VALUE property is a procedure that produces a phrase when evaluated.

Associated with each function is a procedure for combining explanations of the function's inputs into an explanation of its output. The property list of an atom with a functional definition can store a procedure to generate a description of the function's output on the SAY-FUNCTION property.

Since atoms can have both values and functional properties, separate explanations must coexist on the same property list. Consequently there are separate SAY-VALUE and SAY-FUNCTION indicators for each explanation.

The explanation of a function or variable is obtained by evaluating the appropriate explanation property (SAY-FUNCTION or SAY-VALUE). Control of the explanation generator resides in the SAY procedure. To obtain the description of an expression (a variable or functional form), SAY is applied to the expression. Thus,

```
(SAY '(EQUAL (DECK :VALUE)
             (DECK (FGET-VALUE (FGET-VALUE :FRAME 'OBJECT) 'LOCATION))))
```

returns "*the deck of the place is the same as the deck of the location of the object.*"

The SAY algorithm reflects the style of functional decomposition to which LISP lends itself. Tasks are decomposed into subtasks and a function written to combine these partial solutions into the whole. Similarly, the explanation property of function says how to combine the explanations of its arguments into an explanation of the whole. Responsibility for incorporating the explanation of a function's inputs stays within the function. The SAY algorithm needs to select and evaluate the appropriate explanation property depending on whether the LISP statement given it as input is an atom or a functional expression. Also, since the explanation procedure for a particular function call uses the explanations of the inputs, the SAY algorithm must set up a mechanism for refering the function's inputs. SAY's operation is thus analogous to EVAL in LISP.

An explanation generator must obey certain conventions.

We refer to an explanation procedure associated with a particular atom as a generator. The remainder of the explanation process is described in terms of the conventions that a generator must obey.

The output of a generator is a list of tokens to be printed on a terminal. The tokens, when printed, should read as a single phrase (with punctuation included). The phrase may be as short as a single word.

A generator for a function collects the phrases obtained by SAYing each argument of the function and rearranges them into a larger phrase by adding some words of its own to make a readable composite.

A generator does not have to produce the same output each time it is used. A generator can choose to construct its output differently after looking at the number of arguments, at the arguments themselves, or at the phrase produced by SAYing an argument. While a generator can look down as far as it wants into the embedded function calls, it cannot look up to see who is calling it. Sensitivity to global context must be prearranged with other generators by having the superior generator set flags to be sensed by the inferior generator.

Common description generators are predefined.

While the SAY method allows complex generators to be written, many common descriptions are generated simply by interposing a single string with descriptions of the inputs. To accommodate these frequent and simple cases, the following predefined functions (I have used the initial character "*" to indicate SAY functions) can be assigned to the SAY-FUNCTION property of atoms as generators. The comments on the right show the output produced by SAYing instances of a function FN with each of these generators. The lowercase characters (a,b,...,z) in the output are the results of SAYing that input to FN.

```
(*NULFIX string)        ; (FN) => string
(*PREFIX string)        ; (FN A) => string a
(*SUFFIX string)        ; (FN A) => a string
(*INFIX  string)        ; (FN A  B) => a string b
(*INFIXR string)        ; (FN A  B) => b string a
(*INFIXM string)        ; (FN A  B ... Y Z) => a, b, ..., y string z
```

Examples follow to illustrate using these common generators to explain familiar LISP primitives. Each generator would be assigned to the SAY-FUNCTION property of the primitive atom.

```
GENSYM          (*NULFIX '|a new atom|)
CDR             (*PREFIX '|all but the first of|)
BOUNDP          (*SUFFIX '|is bound|)
GREATER         (*INFIX '|is greater than|)
GET             (*INFIXR '|of|)
OR              (*INFIXM '|or|)
```

The following definitions were needed to generate the phrase explaining

SHIPMAINT's preference for the repair place.

EQUAL	(*infix '|is the same as|)	
QUOTE	(sarg 1)	
FGET-VALUE	(cond ((eq (arg 1) ':frame) (cons '|the| (sarg 2))) (t (cons '|the| (*infixr '|of|))))	
DECK	(*prefix '|the deck of|)	
:VALUE	(list '|the| :slot)	*; a SAY-VALUE property.*

Only a few LISP functions are needed to implement SAY.

The following is one implementation of the SAY algorithm, which has already been presented:

```
(defun SAY (e)
       (prog (fn)
         (cond ((atom e)                        ; E is a symbol.
                (setq fn (get e 'SAY-VALUE))
                (and fn (return (eval fn))))
               (t                               ; E is a function.
                (setq fn (get (car e) 'SAY-FUNCTION))
                (and fn (return (apply 'sayl (cdr e))))))
         (return (list e))                      ; E has no defined expansion.
         ))

(defun SAY1 #n
       ;; Generators can now use LEXPR conventions to refer to inputs.
       (eval fn))

(defun SARG (n)
       (SAY (arg n)))
```

From this implementation it can be seen that:

1. The SAY-FUNCTION property of an atom holds the generator for its functional definition.
2. (ARG n) gets the n^{th} argument of the form being said.
3. (SARG n) gets and SAYs the n^{th} argument of the form.
4. *N is the number of arguments in the form.
5. The SAY-VALUE property holds a separate generator for the atom used as a symbol.
6. If the property to say a form is absent, the output of the SAY procedure is a list containing just the form itself.

The common generators are defined as follows:

```
(defun #NULFIX (string)                                  ; (FN) => "string"
       (and string (list string)))

(defun #PREFIX (string)                                  ; (FN A) => "string a"
       (append (and string (list string)) (sarg 1)))

(defun #SUFFIX (string)                                  ; (FN A) => "a string"
       (append (sarg 1) (and string (list string))))

(defun #INFIX (string)                                   ; (FN A B) => "a string b"
       (append (sarg 1) (and string (list string)) (sarg 2)))

(defun #INFIXR (string)                                  ; (FN A B) => "b string a"
       (append (sarg 2) (and string (list string)) (sarg 1)))

(defun #INFIXM (string)                   ; (FN A B ... Y Z) => "a, b,..., y string z"
       (do ((n (- #n 2) (1- n))
            (result (append (sarg (1- #n)) (and string (list string)) (sarg #n))
                    (append (sarg n) result)))
           ((zerop n) result)
           (setq result (cons '/, result))))
```

This implementation of SAY resides in the file RBR; SAY FASL.

Some functions are hard to SAY.

Some programming styles produce functions that are difficult to explain using the SAY method. Since a function's explanation is based on describing the function's effect on its inputs, functions that depend on computed quantities not explicitly passed as arguments effectively hide information essential for clear and complete explanations.

Embedding function calls too deeply can also produce forms that are difficult to explain. Without a way to indicate precedence or otherwise retain the scoping explicitly represented in the s-expression format, the long phrase built up by the SAY method will be ambiguous.

These two restrictions on the suitability of the SAY method for producing explanation of functions might apply with equal force to the human intelligibility of programming styles: implicit arguments and too much functional embedding produce code that is difficult to understand.

The SHOUT function formats the output of SAY.

Proper spacing around punctuation improves the readability of phrases produced by SAY. The SHOUT function takes as input a list of tokens and prints each token, starting a new line as necessary to avoid running off the page. Subsequent lines are indented under the first, although an optioan to SHOUT specifies a column on which to start new lines. SHOUT ordinarily separates tokens by a space, but omits the space before each of { , ; : . ? ! } and puts two spaces after each of { . ? !}. SHOUT omits a space after each of {([{ '}, before each of {)] } '} and around any character that affects only the cursor position. Conflicts between SHOUT's rules for inserting spaces are resolved in favor of the trailing punctuation mark. Note that in order for SHOUT to respond to these punctuation marks, each mark must be an individual token in the input. SHOUT does not inspect tokens longer than a single character.