FUZZY: A PROGRAMMING LANGUAGE FOR FUZZY PROBLEM-SOLVING

WIS-CS-74-202
Computer Sciences Department
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received January 9, 1974

FUZZY: A PROGRAMMING LANGUAGE FOR FUZZY PROBLEM-SOLVING

by

Rick LeFaivre

Technical Report #202

January 1974

FUZZY: A PROGRAMMING LANGUAGE FOR FUZZY PROBLEM-SOLVING

by

Rick LeFaivre

ABSTRACT

A new AI programming language is described which provides facilities for the storage, retrieval and manipulation of fuzzy information. The language (called FUZZY) contains such standard features as an associative data base and a pattern-directed data access and procedure invocation mechanism. These basic facilities are extended, however, by "fuzzifying" the associative net, and by allowing fuzzy procedures and "procedure demons" to be specified for the control of fuzzy processes. The paper discusses some of the major features of FUZZY in the context of previous AI language efforts, presents a simple fuzzy question-answerer as an example, and then summarizes the complete language as it now stands. FUZZY is currently being programmed in LISP on a UNIVAC 1110 computer.

This research was supported in part by a grant from the National Science Foundation (GJ-36312).

INTRODUCTION

As artificial intelligence begins to expand its domain into that mysterious realm we term the "real world," it is clear that we must at some point consider the problem of manipulating fuzzy information. In simple worlds, it may be sufficient to store only exact concepts like (AT FRED (23,76)), or (COLOR BLOCK1 RED). But in more complex worlds--even worlds which do not yet begin to approach the real world in complexity--we must contend with statements like: "I'm fairly certain that FRED is quite close to the center of the room;" or, "BLOCK1 is pretty red, but not as red as BLOCK2." Note that "fuzzy information" is not a particularly well-defined concept. Although the term "fuzzy" has traditionally been associated with the logic of fuzzy sets [8,9], we extend the usage here to include any knowledge which is in some way imprecise, uncertain or probabilistic--in essence, any knowledge which has a numeric weight associated with it, regardless of its semantic interpretation. This paper addresses the problem of representing fuzzy knowledge by describing a LISP-based AI programming language for fuzzy problem-solving (called FUZZY).

The design of FUZZY has been strongly influenced by other AI languages which have appeared in the past few years, particularly MICRO-PLANNER [7], CONNIVER [3,6] and QA4/QLISP [4,5]. Many of the standard features of these languages are present in FUZZY, including an associative data base and a pattern-directed data access and procedure invocation mechanism. A question immediately arises as to whether one of these languages wouldn't be usable, either as-is or in an extended form, for fuzzy problem-solving. For example, CONNIVER and QLISP both allow property lists to be associated with data items, so that a fuzzy truth value could easily be associated with a particular assertion. QLISP, in fact, already uses a MODELVALUE property, although I have not seen any usage of this feature other than for completely "true" or "false" assertions. During 1972 Rob Kling and myself began to discuss some of the modifications which might be made to MICRO-PLANNER to allow it to handle fuzzy problem-solving. The resulting language (FUZZY-PLANNER) was described by Kling in [2], although it was never implemented. Although the purpose of this paper is chiefly to describe FUZZY, not to examine its origins, let me briefly state several reasons why I chose to design a new language rather than making do with an existing system:

(1) I felt that merely tagging fuzzy truth values onto assertions, as could be done in CONNIVER or QLISP, would provide for fuzzy control at only a local level. I desired a more global method of controlling fuzzy processes.

(2) I was not particularly comfortable with any of the existing languages. For example, I desired a control structure which gave the user more direct control over backtracking than MICRO-PLANNER, yet was not as general (or complex) as that of CONNIVER.

(3) Extensive usage of MICRO-PLANNER led me to believe that an efficient, well-conceived implementation is an important goal in the design of a programming language. With this in mind, I wished to design a control structure which did not require a special interpreter, and hence could be embedded directly in LISP (unlike the other AI languages mentioned).

MAJOR FEATURES OF FUZZY

This section describes some of the major features of FUZZY by roughly paralleling the discussion by Bobrow and Raphael in their recent survey article [1]. By so doing, I hope to make it relatively easy to compare FUZZY with the existing AI languages discussed by Bobrow and Raphael. The areas to be covered include:

(1) Data Types and Storage Mechanisms
(2) Control Structures
(3) Pattern-Matching
(4) Deductive Mechanisms

A complete listing of the FUZZY primitives may be found in Appendix B.

DATA TYPES AND STORAGE MECHANISMS

FUZZY is embedded in LISP, and thus allows access to all LISP primitives and data structures. In addition, FUZZY maintains an associative net of assertions similar to that of other AI languages. Any arbitrary LISP list structure may be entered into this net. An assertion, or any sub-part of an assertion, may also have a numeric value associated with it which we will call its fuzzy truth value, or Z-value. Truth values traditionally fall in the range [0,1], although in FUZZY other ranges may be used if desired (the lower and upper bounds of the Z-value range are specified by the LISP variables ZLOW and ZHIGH). FUZZY makes no assumptions about the interpretation given to a particular Z-value, although again it is traditional to equate ZLOW with "false" and ZHIGH with "true". Internally, Z-values are stored by CONSing (appending) the numeric value to the assertion or sub-assertion modified. Thus

```
[(MADE-OF MOON CHEESE) . 0]
[(CERTAIN JOHN [(PRETTY MARY) . 0.9] . 0.6]
```

are both valid fuzzy assertions. (The second example might be interpreted "John is relatively (0.6) certain that Mary is very (0.9) pretty."). Assertions or sub-assertions which have no fuzzy truth value associated with them are given a default Z-value (ZHIGH) by the system.

Standard primitives are available for accessing this fuzzy associative net. For example,

```
(ADD (PROUD MARY) 0.9)
```

adds the assertion [(PROUD MARY) . 0.9] to the net, and

```
(REMOVE (PROUD MARY))
```

removes it. The statements

```
(FETCH (PRETTY ?X)) or (FETCH (PRETTY ?X) [ZLOW,ZHIGH])
(FETCH (PRETTY ?X) 0.9) or (FETCH (PRETTY ?X) [0.9,ZHIGH])
(FETCH (PRETTY ?X) [0.0,0.5])
```

all retrieve something which is PRETTY within a specified range, binding the object to the FUZZY variable ?X (the pattern-matcher will be discussed in a later section). When a FETCH-pattern matches several assertions with Z-values in the proper range, the one with the highest Z-value is normally returned. However, by reversing the order of the range elements, the assertion with the lowest Z-value in the range may be requested. For example,

```
(FETCH (PRETTY ?X) [0.5,0.0])
```

will return the most "unpretty" object in the net.

A data context mechanism is also available in FUZZY via the SAVE and RESTORE primitives. SAVE saves the current associative net, along with any variables specified, and RESTORE restores things to a previous state. For example,

```
(CSETQ STATE (SAVE !X Y))
.
.
.
(RESTORE STATE)
```

will save and then restore the associative net, the FUZZY value of X, and the LISP value (and property list) of Y. As will be seen in the next section, there are several situations where FUZZY automatically saves and then restores the net, relieving the user of this responsibility.

CONTROL STRUCTURES

FUZZY provides two major types of program control: a rather standard fail-succeed mechanism with user-controlled backtracking and state restoration capabilities; and a fuzzy control structure which appears to be unique to FUZZY. Each FUZZY statement returns both a value and a fuzzy truth value--the Z-value is CONSed to the

value as with fuzzy assertions. The primitives VAL and ZVAL return the value and Z-value portions respectively (a default value (ZHIGH) is returned by ZVAL if no truth value is present). If VAL or ZVAL is called with no arguments, the value or Z-value portion of the last FETCH (or GOAL) expression is returned. Thus

```
(ZVAL (FETCH (RED BLOCK1)))
```

and

```
(FETCH (RED BLOCK1))
(ZVAL)
```

both return the same Z-value.

If the value portion of a result is the atomic symbol *FAIL, the original expression is said to have failed (*FAIL, which is bound to itself, is thus similar to NIL in LISP and MICRO-PLANNER). Any other value causes the expression to succeed.

Local Success-Failure Control

Local control of success and failure is provided via the several if-statements in FUZZY:

```
(IF <exp> THEN: <s1> <s2> ... ELSE: <f1> <f2> ...)
```

evaluates the "THEN:" or "ELSE:" expressions as a function of whether <exp> succeeds or fails (a value of NIL is interpreted as failure in this case so that standard LISP predicates may be used); IFALL takes a number of <exp>s, and requires all of them to succeed (actually, IF is identical to IFALL); and IFANY is similar to IFALL, except it requires only one of the <exp>s to succeed.

Local Fuzzy Control

FUZZY equivalents of the standard AND, OR and NOT predicates are available as ZAND, ZOR and ZNOT. ZAND returns a Z-value equal to the minimum of those of its arguments, and ZOR the maximum. ZAND and ZOR may also be given a threshold, such that if any Z-value falls below the threshold the ZAND or ZOR immediately fails. Using the standard [0,1] Z-value range, ZNOT simply returns a Z-value of one minus that of its argument. This may be changed for other Z-value ranges, e.g., the negation of its argument for the range [-1,+1].

Backtrack Control

Following the lead of Sussman and McDermott [6], backtracking in FUZZY is under the direct control of the user. This seems to provide for more efficient, controllable and understandable programs, as well as allowing for a much more efficient implementa-

tion of the language. In addition to allowing the user to construct his own backtrack mechanisms using the SAVE and RESTORE context primitives discussed earlier, FUZZY provides several system backtrack primitives: the FOR statement iterates through an explicit list of alternatives in a manner similar to THAMONG in MICRO-PLANNER; the FOREACH statement iterates through all assertions in the net (obtainable via the FETCH primitive) which match a given pattern; and the FORALL statement iterates through all assertions obtainable via the GOAL primitive (to be discussed under Deductive Mechanisms) which match a given pattern. For example, each of the following will print all blocks which are either very pretty, very red or very big:

```
(FOREACH (BLOCK ?X)
 (IFANY (FETCH (PRETTY !X) 0.9)
        (FETCH (RED !X) 0.9)
        (FETCH (BIG !X) 0.9)
  THEN: (PRINT !X)))

(FOREACH (BLOCK ?X)
 (FOR ?REL (PRETTY RED BIG)
  (FETCH (!REL !X) 0.9)
  (PRINT !X)))

(FOR ?REL (PRETTY RED BIG)
 (FOREACH (!REL ?X) ZVAL: 0.9
  (FETCH (BLOCK !X))
  (PRINT !X)))
```

Upon entry to one of the for-statements, the current state of the associative net and the "accumulated Z-value" discussed in the next section are saved for backtrack usage. If any of the expressions within the for-statement fail, or if (BACK) is evaluated, the net and accumulated Z-value are automatically restored and the next alternative is tried. The next alternative may also be tried without backtracking via (NEXT), or the loop may be exited via (EXIT). A default (NEXT) is placed at the end of the loop, so that:

```
(FOREACH (RED ?X) (REMOVE (RED !X)))
```

will indeed remove all the red objects from the net, whereas:

```
(FOREACH (RED ?X)
 (REMOVE (RED !X))
 (BACK))
```

would have no effect because of the backtracking. Note that FOREACH automatically orders the alternatives in increasing or

decreasing order by Z-value, and that the value and Z-value portions of the assertions retrieved are available via the expressions (VAL) and (ZVAL) until the next FETCH or GOAL expression.

Fuzzy Procedures

Up to this point we have outlined several of the forms of local control which are available in FUZZY: success or failure of individual statements or groups of statements may be monitored via VAL, IF, IFALL and IFANY; fuzzy truth values may be manipulated via ZVAL, ZAND, ZOR and ZNOT; and iteration through a set of alternatives with user-controlled backtracking is available via FOR, FOREACH and FORALL. The next step is to specify a general formalism for combining these various primitives into units of procedural knowledge, or procedures.

A FUZZY procedure is much like a MICRO-PLANNER theorem or QLISP QLAMBDA function: it takes a single argument which is matched against a procedure pattern, the procedure is entered, and a result is computed. For example, the following simple procedure reverses an ordered pair, i.e., (SWITCH (A B)) returns (B A):

```
(PROC NAME: SWITCH (?X ?Y) (SUCCEED (!Y !X)))
```

The "NAME:" field is optional--if absent a unique name will be generated by the system (this name is returned as the value of the PROC statement). Procedures may either succeed (returning both a value and Z-value) or fail, just like standard system primitives. Success is caused by (SUCCEED <val> <zval>) or simply (SUCCEED), which returns the instantiated procedure pattern with the "accumulated Z-value" discussed below. Failure may be caused by simply (SUCCEED *FAIL). However, in this case we typically want to restore the net to its state before the procedure was entered, which may be accomplished via (RESTORE). Thus we normally do (FAIL), which is equivalent to (RESTORE) followed by (SUCCEED *FAIL).

It should be noted that FUZZY operates in a kind of "inverse declaration mode"--all variables are assumed to be local to the procedure in which they appear unless explicitly declared global, e.g.:

```
(PROC GLOBAL: (!X !Y) ...)
```

Since in most cases variables are used only locally, this relieves the user of much of the burden of making variable declarations.

Global Control

The major difference between FUZZY procedures and units of

procedural knowledge in other programming languages is in the amount of global control a procedure exercises over its local computations. For example, LISP functions and PROGs exercise no global control--all decisions (e.g., when to exit from the function or PROG) are made at a local level by statements within the routine. On the other hand, MICRO-PLANNER monitors the execution of its theorems, looking for statements which fail (return NIL) and taking some global action when necessary (backtracking or causing the theorem to fail). We are faced with the question of what form of global control (if any) should be built into a FUZZY procedure. Consider a typical example: we want a procedure to succeed only if each of its local statements succeeds with a Z-value above some threshold value. Now, this could of course be done with only local control:

```
(PROC <pat>
      (IF (LT (ZVAL <e1>) THRESH) THEN: (FAIL))
      (IF (LT (ZVAL <e2>) THRESH) THEN: (FAIL))
      .
      .
      .
      (SUCCEED))
```

but this becomes quite messy. We would like to just specify the threshold value and have the system perform the necessary checking:

```
(PROC THRESH: <n> <pat> <e1> <e2> ...)
```

A fixed procedure mechanism of this form would be the FUZZY equivalent of theorems in MICRO-PLANNER. However, there are other forms of global control which might be desired--for example: ignore all failures; succeed only if at least n of the statements succeed; succeed only if some function of the individual truth values exceeds a threshold (e.g., a threshold operator); etc. FUZZY procedures allow all of these forms of global control (and more) via the specification of procedure demons.

Procedure Demons

A procedure demon is a LISP function which is associated with a procedure, and which is given control after each "interruptable statement" of the procedure is evaluated. (An interruptable statement is a statement which returns a value [e.g., a GOTO is not interruptable], and which occurs either at the top level of the procedure, in the "THEN:" or "ELSE:" portions of an if-statement, or in a for-statement [the initial implied FETCH or GOAL of a FOREACH or FORALL statement is considered to be an interruptable statement].) The demon is passed the result of the evaluation, the threshold value associated with the procedure, and

an "accumulated Z-value" which may be dynamically computed by the demon. For example, consider the case of succeeding only if all the interruptable statements in a procedure exceed a certain threshold. A procedure demon which exercises this form of global control might be defined as follows (this is in fact FUZZY's standard default demon):

```
(CSETQ DEMON1
(LAMBDA (D-VAL D-THRESH D-ACCUM)
        (COND [<EQ (VAL D-VAL) *FAIL> <FAIL>]
              [<LT (ZVAL D-VAL) D-THRESH> <FAIL>]
              [<LT (ZVAL D-VAL) D-ACCUM> <ZVAL D-VAL>]
              [T D-ACCUM])))
```

Note that the value returned by a procedure demon is saved by the system and becomes the new accumulated Z-value. Thus in addition to checking for statements which fail or fall below the threshold, DEMON1 keeps track of the lowest Z-value encountered. This value may ultimately be returned as the truth value of the procedure call via (SUCCEED). A procedure which uses DEMON1 might be defined as follows:

```
(PROC DEMON: DEMON1 THRESH: 0.5 ACCUM: 1.0
      <pat> <e1> <e2> ...)
```

This indicates that the demon for this procedure is DEMON1, the threshold is 0.5, and the initial accumulated Z-value is 1.0. The "DEMON:", "THRESH:" and "ACCUM:" fields are all optional, with standard default values used if omitted. Various other standard procedure demons are supplied by the system, or the user may write his own to specify unique forms of global control.

PATTERN-MATCHING

FUZZY's pattern-matching capabilities are similar to those of QLISP, although FUZZY offers more built-in pattern functions. Like other AI languages, FUZZY operates in "inverse quote mode", i.e., constants stand for themselves and variables and expressions are flagged as such. However, in FUZZY this concept is extended to include functional arguments as well as patterns. Each primitive function specifies whether its arguments are to be <u>evaluated</u> in the normal LISP sense, or <u>instantiated</u>, with flagged variables and expressions replaced by their values. The user may overide the default interpretations via use of the instantiation operator (!) or evaluation operator (&). The standard LISP QUOTE operator (') is also available to overide both instantiation and evaluation. For example, the primitive SUCCEED specifies that its first argument be instantiated and its second argument be evaluated.

Thus

```
(SUCCEED (BIG !OB) (TIMES (ZVAL) !BIAS))
```

may be written instead of

```
(SUCCEED !(BIG !OB) &(TIMES (ZVAL) !BIAS))
```

However, one may overide the default interpretation:

```
(SUCCEED &(CAR L) !X)
```

The "!" and "&" operators may also be used within patterns, e.g., (A !X &Y) instantiates to (A B C) if X has FUZZY value B and Y has LISP value C. "Segment" instantiation and evaluation operators are also available for use within patterns, e.g., if X has FUZZY value (B C) and Y has LISP value (D E), (A !!X &&Y) instantiates to (A B C D E) [whereas (A !X &Y) would instantiate to (A (B C) (D E))].

FUZZY variables may be assigned values via the "?" (item) and "??" (segment) operators. For example, using the primitive MATCH:

```
(MATCH (?X ??Y) (A B C))
```

binds X to A and Y to (B C). "?" and "??" may also stand alone to match an item or segment with no variable assignments. Note that in addition to MATCH, which instantiates both of its arguments, there is a more conventional BIND primitive which <u>evaluates</u> its second argument. Thus

```
(BIND ?X (CAR L))
```

is equivalent to

```
(MATCH ?X &(CAR L))
```

Several additional features of FUZZY's pattern-matcher are illustrated in the following examples:

<u>TIC-TAC-TOE Example</u>

Consider the problem of checking whether a TIC-TAC-TOE board stored in the form

```
((X O -) (O X -) (X O X))
```

is a winning board for X. A LISP expression to do this might look like:

```
(OR [MEMBER '(X X X) BOARD]
    [LINE CAAR CAADR CAADDR]
    [LINE CADAR CADADR CADADDR]
    [LINE CADDAR CADDADR CADDADDR]
    [LINE CAAR CADADR CADDADDR]
    [LINE CADDAR CADADR CAADDR])
```

where LINE is an auxiliary function:

```
(CSETQ LINE
(LAMBDA (F1 F2 F3)
        (AND [EQ (F1 BOARD) 'X]
             [EQ (F2 BOARD) 'X]
             [EQ (F3 BOARD) 'X])))
```

Although this is relatively compact, the necessary CAADR's and CADDADDR's make it a headache both to program and interpret. The corresponding FUZZY pattern, however, is quite straightforward:

```
(*OR [?? (X X X) ??]
     [(*REP (X ? ?) 3)]
     [(*REP (? X ?) 3)]
     [(*REP (? ? X) 3)]
     [(X ? ?) (? X ?) (? ? X)]
     [(? ? X) (? X ?) (X ? ?)])
```

*OR and *REP are pattern functions, *OR signifying that only one of its arguments must match, and *REP that its pattern is to be repeated n times.

Pattern Recognition Example

As another example, consider a pattern recognition program where a grid is stored as a list of rows, e.g., "I" might appear as:

```
((1 1 1 1 1)
 (0 0 1 0 0)
 (0 0 1 0 0)
 (0 0 1 0 0)
 (1 1 1 1 1))
```

We wish to scan a 3 x 3 characterizer, e.g.,

```
((- 1 -)
 (0 1 0)
 (1 1 1))
```

across the grid, noting the row and column position of its upper left corner if it matches. This would require a fairly complex

LISP program, but in FUZZY we merely define the characterizer as:

```
(??/ROW (??/COL        ? 1 ? ??)
        ((*REP ? !COL) 0 1 0 ??)
        ((*REP ? !COL) 1 1 1 ??) ??)
```

Here ??/X (like ??X and ??), matches a segment of zero or more items, but assigns the length of the segment matched to the FUZZY variable X.

DEDUCTIVE MECHANISMS

The deductive capabilities of FUZZY are similar to those of MICRO-PLANNER. In addition to ADD, REMOVE, FETCH and FOREACH, which operate only on the associative net, FUZZY supports ASSERT, ERASE, GOAL and FORALL primitives, which can call procedures via a standard pattern-directed invocation mechanism. Such procedures are added to the data base via (ADD <type> <proc>), where <type> is "ASSERT:", "ERASE:" or "GOAL:", and <proc> is a procedure name or a PROC statement. For example,

```
(ADD GOAL: (PROC (DISHONEST ?X)
                 (FETCH (TRICKY !X))))
```

establishes a "goal procedure" which says in effect that all tricky people are dishonest. If (GOAL (DISHONEST DICK)) is ever evaluated and (DISHONEST DICK) is not stored explicitly in the net, the above procedure will be called. The original GOAL statement will then succeed if (TRICKY DICK) can be found in the net, returning a Z-value equal to that of (TRICKY DICK).

Like MICRO-PLANNER, FUZZY must also contend with the more complex case where the GOAL pattern is not fully instantiated, e.g., (GOAL (DISHONEST ?)). The ?X in the procedure would then be unbound, and (TRICKY !X) would not be instantiated properly. As might be expected, FUZZY handles this situation by treating the "!" (or "!!") operator as identical to the "?" (or "??") operator when applied to an unbound variable. A tricky person would thus be retrieved from the net and returned to the original GOAL statement as being dishonest.

Finally, consider the case where a goal procedure is activated by a FORALL statement (FORALL is identical to FOREACH, except that FORALL uses GOAL to retrieve assertions which match the pattern, whereas FOREACH uses FETCH):

```
(FORALL (DISHONEST ?Y)
        (PRINT !Y))
```

This is exactly the case where we want our goal procedure to act like a generator in the CONNIVER sense. We would like it to iterate through all of the dishonest people it knows about, which in this case are all tricky people, returning one at a time until the original FORALL exits from its loop. This may be accomplished in FUZZY via the SUCCEED? statement. SUCCEED? is identical to SUCCEED, except that it saves the current state of the procedure so that it can be restarted by a higher-level FORALL if necessary. Our example now becomes:

```
(ADD GOAL: (PROC (DISHONEST ?X)
                 (FOREACH (TRICKY !X)
                  (SUCCEED?))))
```

When called from a GOAL statement, this procedure will return the first tricky person found as being dishonest just as in the original version. When invoked by a FORALL statement, however, the procedure will act like a generator, returning one tricky person at a time until it runs out or the FORALL is satisfied. It should be noted that the state suspension and restoration mechanism described here is not nearly as general as that of CONNIVER. It does allow generators to be written however, and by restricting this capability somewhat a much more efficient implementation of the language is made possible.

CONCLUDING REMARKS

FUZZY is a first attempt at combining some of the Good Ideas which have appeared in AI languages over the past few years with a usable method of accessing and manipulating fuzzy knowledge. The major design goal was that of attaining an efficient, usable, yet powerful fuzzy programming language. The decisions to allow only recursive backtracking and to restrict the generality of the state suspension and restoration mechanism have resulted in an efficient implementation which does not require the extensive control tree maintenance necessary in other AI languages. In particular, unlike MICRO-PLANNER and CONNIVER, a special interpreter is not required--FUZZY is embedded directly in LISP, and LISP and FUZZY primitives may be freely intermixed. This ability to invoke FUZZY primitives from LISP functions (which may be compiled) allows the user to write far more efficient programs than would be possible in a language like MICRO-PLANNER.

One of the major goals of artificial intelligence research is the search for powerful and efficient representations for knowledge. Certain types of knowledge are best represented in traditional forms, e.g., as strings, arrays or lists. In other situations net structures which are accessed associatively seem to provide more power and flexibility. It is now becoming apparent that much knowledge is of sufficient complexity to be best represented procedurally. One of the major contributions of the new AI languages is a shift in viewpoint from the programming language as a manipulator of knowledge, to the programming language as a representation of knowledge. With this view in mind, it is my hope that FUZZY will allow us to begin to consider the problem of representing the gray, along with the black and the white.

REFERENCES

[1] Bobrow, D. and B. Raphael, "New Programming Languages for AI Research," SRI AI Center Technical Note 82, August 1973.

[2] Kling, R., "Fuzzy Planner: Computing Inexactness in a Procedural Problem-Solving Language," Technical Report No. 168, Computer Sciences Department, University of Wisconsin, February 1973.

[3] McDermott, D. and G. Sussman, "The CONNIVER Reference Manual," MIT AI Memo No. 259, May 1972.

[4] Reboh, R. and E. Sacerdoti, "A Preliminary QLISP Manual," SRI AI Center Technical Note 81, August 1973.

[5] Rulifson, J., J. Derksen and R. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," SRI AI Center Technical Note 73, November 1972.

[6] Sussman, G. and D. McDermott, "Why Conniving is Better Than Planning," MIT AI Memo No. 255A, April 1972.

[7] Sussman, G., T. Winograd and E. Charniak, "MICRO-PLANNER Reference Manual," MIT AI Memo No. 203A, December 1971.

[8] Zadeh, L., "Fuzzy Sets," Information and Control, vol. 8, 338-353, June 1965.

[9] Zadeh, L., "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes," IEEE Transactions on Systems, Man and Cybernetics, vol. SMC-3, pp. 28-44, January 1973.

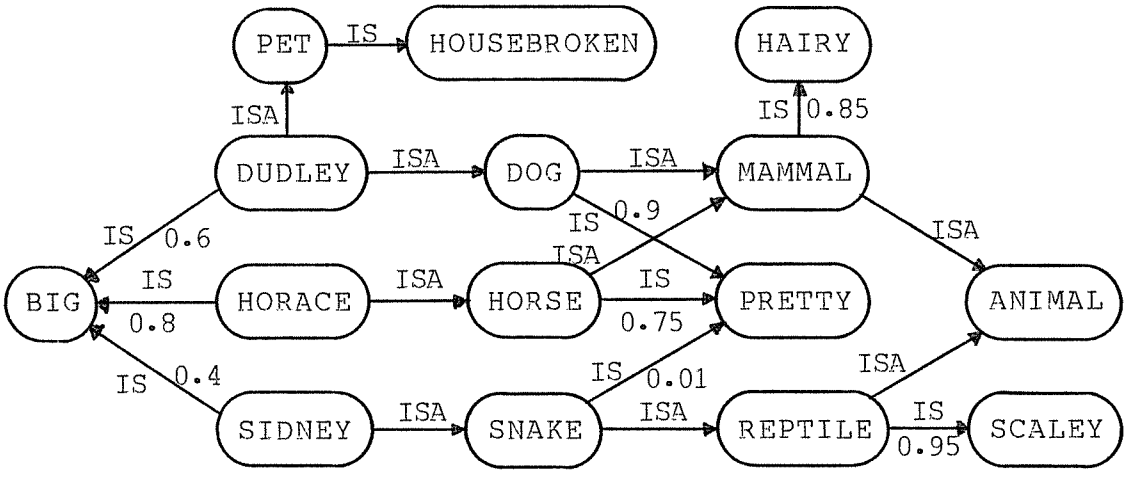
APPENDIX A: A SAMPLE FUZZY PROGRAM

In this appendix a simple question-answerer is presented which illustrates some of the features available in FUZZY. The program answers questions about the information stored in a semantic network built out of the relations ISA (class membership) and IS (fuzzy set membership). ISA is transitive, so that if (HORACE ISA HORSE) and (HORSE ISA MAMMAL), then (HORACE ISA MAMMAL). Similarly, IS is transitive over ISA, e.g., if (DUDLEY ISA PET) and (PET IS HOUSEBROKEN), then (DUDLEY IS HOUSEBROKEN). IS is also a fuzzy relation--dogs and horses may both be pretty, but dogs might be considered prettier than horses. (Note that additional relations could be added to the net if desired with very little change to the program).

A net of sufficient complexity to illustrate the workings of the program may be generated by the following FUZZY statements:

```
(ADD (HORACE ISA HORSE))          (ADD (DOG ISA MAMMAL))
(ADD (HORACE IS BIG) 0.8)         (ADD (DOG IS PRETTY) 0.9)
(ADD (DUDLEY ISA DOG))            (ADD (SNAKE ISA REPTILE))
(ADD (DUDLEY ISA PET))            (ADD (SNAKE IS PRETTY) 0.01)
(ADD (DUDLEY IS BIG) 0.6)         (ADD (PET IS HOUSEBROKEN))
(ADD (SIDNEY ISA SNAKE))          (ADD (MAMMAL ISA ANIMAL))
(ADD (SIDNEY IS BIG) 0.4)         (ADD (MAMMAL IS HAIRY) 0.85)
(ADD (HORSE ISA MAMMAL))          (ADD (REPTILE ISA ANIMAL))
(ADD (HORSE IS PRETTY) 0.75)      (ADD (REPTILE IS SCALEY) 0.95)
```

Pictorially, the resulting net looks like:



The program, which may be found at the end of this appendix, answers questions of the form "Is an X a Y" (X ISA Y) or "Is an X Y" (X IS Y). The answers generated by the program are somewhat more complex, attempting to give additional information when relevant. For example, consider the following sample dialogue:

```
(QA)
  HI
(IS DUDLEY A MAMMAL)
  (YES, DUDLEY IS A MAMMAL)
(IS HORACE PRETTY)
  (YES, HORACE IS RELATIVELY PRETTY)
(IS AN ANIMAL HAIRY)
  (POSSIBLY: MAMMAL IS VERY HAIRY)
(IS HORACE SCALEY)
  (DON'T THINK SO, BUT REPTILE IS EXTREMELY SCALEY)
(IS SIDNEY BIG)
  (NO, SIDNEY IS NOT VERY BIG)
(IS A MAMMAL HOUSEBROKEN)
  (POSSIBLY: DUDLEY IS HOUSEBROKEN)
(IS A SNAKE PRETTY)
  (NO, SNAKE IS NOT PRETTY)
(IS A SNAKE A MAMMAL)
  (DON'T THINK SO, BUT DOG IS A MAMMAL)
(IS AN ANIMAL A PET)
  (POSSIBLY: DUDLEY IS A PET)
(IS DUDLEY A DADDY)
  DON'T KNOW
(WHAT IS LIFE)
  HUH?
STOP
  GOODBYE
```

Considering the simplicity of the program, these responses appear to be relatively sophisticated. Obviously, small changes could be made to make the output more stylized--for example, nouns could be checked to see whether an "A", "AN" or nothing should proceed them. More important, however, is the ease with which variant question and answer forms may be added to the program. Each valid input form is represented by a procedure which translates the input into an internal representation--for example, TRANS1 transforms (IS DUDLEY A MAMMAL) into (DUDLEY ISA MAMMAL). The various answer types are also represented by procedures. To add a new input or output form, one merely defines the proper procedure and adds it to the TRANSTYPES or ANSTYPES list. For example, suppose we want to handle questions of the form (WHAT IS HAIRY) or (WHO IS A DOG). Note first that fill-in-the-blank questions can already be asked, although the question and answer are not in the desired format:

```
(IS WHAT HAIRY)
  (DON'T THINK SO, BUT MAMMAL IS VERY HAIRY)
```

To handle such questions directly, we need only do the following:

```
(PUSH TRANSTYPES
(PROC ((*OR WHAT WHO) IS (*OR A AN) ?Y)
      (SUCCEED (ISA !Y))))
(PUSH TRANSTYPES
(PROC ((*OR WHAT WHO) IS ?Y)
      (SUCCEED (IS !Y))))
(PUSH ANSTYPES
(PROC (?R ?Y)
      (GOAL (?Z !R !Y) 0.5)
      (SUCCEED (!Z "IS" &&(ADJ (ZVAL)) !Y))))
```

Other more complex question types (e.g., conjunctions and disjunctions) could be handled in a similar manner.

Several points should be made about the type of information which appears in the semantic net. Note that the present net specifies that all mammals are very hairy. Consider adding an exception to the general rule, however: whales are mammals which are not hairy. One solution to this problem is to remove the hairiness attribute from the class of mammals and add it to each individual mammal which is hairy. However, a more elegant solution is to simply add exceptions directly to the net, e.g.:

```
(ADD (WHALE ISA MAMMAL))
(ADD (WHALE IS HAIRY) 0.0)
```

All mammals but whales would then retain their hairiness. Making use of the deductive apparatus of the system in this way allows the user to keep the number of relations which must be stored explicitly to a minimum.

Consider also the problem of making ISA a fuzzy relation like IS. For example, we may not be too sure of SIDNEY's heritage--we think he is a snake, but it is also possible that he is a lizard. Due to the presence of procedure demons, the present program will handle fuzzy ISAs without modification. When making deductions over a chain of ISAs, the Z-value of the deduction will automatically be bounded by the minimum of the Z-values of the individual relations (recall the standard default demon discussed earlier). To specify other forms of control over such fuzzy chains of inference--perhaps the average of the individual Z-values--we need only change the procedure demon. This flexibility at least hints at the power of the procedure demon fuzzy control structure.

Finally, note that the programmer has complete control over

the way in which searches are made through the net (without having to explicitly program them). For example, the transitivity procedure for ISA works fine when searching in the "forward" direction, e.g.,

```
(GOAL (HORACE ISA ?))
```

Consider, however, searching backwards through the net as is done by the procedure ANS2:

```
(FORALL (?Z ISA !X) ...)
```

The given transitivity procedure will handle this case, but it will thrash around a lot before it hits upon the correct path. If we are concerned about efficiency (as we certainly would be with a larger data base) we might consider breaking the transitivity mechanism for ISA into two more efficient sub-cases:

```
(ADD GOAL: (PROC ((*INST ?X) ISA ?Y)
                 (FOREACH (!X ISA ?Z)
                  (FORALL (!Z ISA !Y)
                   (SUCCEED?)))))
(ADD GOAL: (PROC (?X ISA (*INST ?Y))
                 (FOREACH (?Z ISA !Y)
                  (FORALL (!X ISA !Z)
                   (SUCCEED?)))))
```

The search would then start at the node which is given and proceed forward or backward as necessary.

This simple example was intended both to demonstrate the structure of a complete FUZZY program and to hint at what might be done in a more realistic setting. Fuzziness was used here in only a superficial way. A more powerful question-answerer would almost certainly allow fuzzy questions as input ("What is very pretty, relatively big and is housebroken?"--"Why, Dudley the dog, of course!"). Fuzzy problem-solving techniques also allow learning to take place in a natural way--tentative facts can be added to the net with relatively low Z-values, and then modified as a function of subsequent information. We might also consider other uses for Z-values, e.g., as strength indicators which allow the program to wander through a fuzzy net, generating pertinent responses during a "fuzzy conversation." The point to be made is that FUZZY allows the program designer to think about problems like this without having to worry about lower-level details like storing and retrieving fuzzy data, searching through a fuzzy associative net, or iterating through sets of fuzzy alternatives.

```
;                A SIMPLE FUZZY QUESTION-ANSWERER
;************************** MONITOR **************************
(CSETQ QA (LAMBDA NIL
 (PROG  (QUES)
        (PRINT "HI")
  LOOP: (IF [EQ (SETQ QUES (READ)) 'STOP]
         THEN: (RETURN "GOODBYE"))
        (IF [SETQ QUES (TRY &TRANSTYPES &QUES)]
         THEN: (IF [SETQ QUES (TRY &ANSTYPES &QUES)]
                THEN: (PRINT (VAL QUES))
                ELSE: (PRINT "DON'T KNOW"))
         ELSE: (PRINT "HUH?"))
        (GO LOOP:))))
;************ DATA BASE (TRANSITIVITY) PROCEDURES ************
(ADD GOAL: (PROC (?X ISA ?Y)
                 (FOREACH (!X ISA ?Z)
                  (FORALL (!Z ISA !Y)
                   (SUCCEED?)))))
(ADD GOAL: (PROC (?X IS ?Y)
                 (FORALL (!X ISA ?Z)
                  (FORALL (!Z IS !Y)
                   (SUCCEED?)))))
;***************** INPUT TRANSLATION PROCEDURES ****************
(PROC NAME: TRANS1 (IS &ARTICLE ?X (*OR A AN) ?Y)
                   (SUCCEED (!X ISA !Y)))
(PROC NAME: TRANS2 (IS &ARTICLE ?X ?Y)
                   (SUCCEED (!X IS !Y)))
(CSETQ ARTICLE '(*OPT (*OR A AN)))
(CSETQ TRANSTYPES '(TRANS1 TRANS2))
;***************** ANSWER-GENERATING PROCEDURES ****************
(PROC NAME: ANS1 (?X ?R ?Y)
  (GOAL (!X !R !Y))
  (BIND ?ANS (IF [GE (ZVAL) 0.5] THEN: "YES," ELSE: "NO,"))
  (SUCCEED (!ANS !X "IS" &&(ADJ (ZVAL)) !Y)))
(PROC NAME: ANS2 (?X ?R ?Y)
  (FORALL (?Z ISA !X)
   (IF [GOAL (!Z !R !Y) 0.5] THEN: (EXIT) ELSE: (BACK))
  (SUCCEED ("POSSIBLY:" !Z "IS" &&(ADJ (ZVAL)) !Y)))
(PROC NAME: ANS3 (? ?R ?Y)
  (GOAL (?Z !R !Y))
  (SUCCEED ("DON'T THINK SO, BUT" !Z "IS" &&(ADJ (ZVAL)) !Y)))
(CSETQ ANSTYPES '(ANS1 ANS2 ANS3))
;******************** ADJECTIVE GENERATOR ********************
(CSETQ ADJ (LAMBDA (Z)
 (IF [EQ !R 'ISA] THEN: '(A)
  ELSE: (FOR ?PAIR &ADJECTIVES
         (IF [GE Z (CAR !PAIR)] THEN: (EXIT (CDR !PAIR)))))))
(CSETQ ADJECTIVES '((1.0) (0.95 EXTREMELY) (0.8 VERY)
          (0.5 RELATIVELY) (0.1 NOT VERY) (0.0 NOT)))
```

APPENDIX B: FUZZY PRIMITIVES

This appendix briefly describes each of the primitives currently available in FUZZY. The complete calling sequence for each primitive is given, with <pat> and <skel> indicating arguments which will be instantiated by the system. All other arguments will be evaluated unless indicated otherwise. Default values for missing arguments are indicated where appropriate. Note that the default value for a missing <zval> argument is always ZHIGH, and the default value for a missing <zrange> argument is always [ZLOW,ZHIGH].

FUNCTIONS:

(ADD <skel> <zval>) or (ADD <type> <proc>)

Adds (<skel> . <zval>) to the associative net. If the first argument is "GOAL:", "ASSERT:" or "ERASE:", the second argument should be a procedure name or an expression which evaluates to a procedure name (usually a PROC statement).

(ASSERT <skel> <zval>)

Adds (<skel> . <zval>) to the net and calls all assert procedures which match <skel>. Fails if any of the assert procedures fail (in which case <skel> is removed).

(BACK <val> <zval>)

Causes the current for-statement to backtrack and try the next alternative. If no alternatives remain, a value of (<val> . <zval>) is returned as the value of the for-statement. Default <val> is *FAIL.

(BIND <pat> <exp>)

Matches <pat> (usually ?<name>) against the value of <exp>.

(BOUND <var>)

Fails unless the FUZZY variable <var> (?<name> or !<name>) is currently bound.

(ERASE <skel>)

Removes <skel> from the net and calls all erase procedures which match <skel>. Fails if <skel> is not in the net, or if any of the erase procedures called fail (in which case <skel> is restored).

(EXIT <val> <zval>)

Exits from the current for-statement with a value of (<val> . <zval>). Default <val> is the instantiated for-pattern.

(FAIL <context>)

Executes (RESTORE <context>) followed by (SUCCEED *FAIL).

(FETCH <pat> <zrange>)

Fetches an assertion from the net matching <pat> whose Z-value is in the proper range. <zrange> may be either [<lower>,<upper>] (return assertion with highest Z-value), [<upper>,<lower>] (return assertion with lowest Z-value), or <lower> (assume an <upper> of ZHIGH). Default <zrange> is [ZLOW,ZHIGH].

(FLUSH <flag>)

(FLUSH) removes all assertions from FUZZY's associative net. (FLUSH T) also removes all procedures.

(FOR <pat> <list> <e1> <e2> ...)

<list> should instantiate to a list. <pat> will be matched successively against each item of <list> and, for each successful match, the <e>s will be evaluated. Backtracking upon failure or (BACK).

(FORALL <pat> ZVAL: <zrange> <e1> <e2> ...)

Iterates through all assertions obtainable via GOAL which match <pat> and have a Z-value in the proper range. Backtracking upon failure or (BACK). See FETCH for a description of the <zrange> parameter.

(FOREACH <pat> ZVAL: <zrange> <e1> <e2> ...)

Same as FORALL except iterates only through assertions obtainable via FETCH.

(GOAL <pat> <zrange>)

First performs a FETCH--if unsuccessful calls relevant goal procedures until one succeeds.

(GOTO <tag>)

Transfers control to <tag> in the current procedure. <tag> is evaluated if it is non-atomic.

(IF <exp> THEN: <s1> <s2> ... ELSE: <f1> <f2> ...)

Evaluates <f1> <f2> ... if <exp> returns *FAIL or NIL, otherwise evaluates <s1> <s2> ... Value is the value of the last expression evaluated. If the "THEN:" is missing and <exp> succeeds, the value of <exp> is returned. If the "ELSE:" is missing, "ELSE: *FAIL" is assumed.

(IFALL <e1> <e2> ... THEN: <s1> <s2> ... ELSE: <f1> <f2> ...)

Similar to IF, except all of the <e>s must succeed.

(IFANY <e1> <e2> ... THEN: <s1> <s2> ... ELSE: <f1> <f2> ...)

Similar to IFALL, except only one of the <e>s need succeed.

(MATCH <pat> <skel>)

Matches <pat> against <skel>. Succeeds (with value <skel>) only if the match succeeds.

(NEXT <val> <zval>)

Causes the current for-statement to try the next alternative (like BACK), except no backtracking is performed. Default <val> is the instantiated for-pattern.

(NOHASH <at1> <at2> ...)

The indicated atoms will not be hashed into the associative net. Saves space and time for heavily-used but not significant atoms.

(POP <var>)

<var> should be a LISP or FUZZY variable which is bound to a list. The first member of the list will be removed.

(PROC NAME: <name> GLOBAL: <list> DEMON: <name> THRESH: <n>
ACCUM: <n> <pat> <e1> <e2> ...)

Defines a procedure and returns its name. The "NAME:" through "ACCUM:" fields are all optional--a unique name will be generated by the system and a standard default demon will be used if missing. <list> should be a list of FUZZY variables which are global to this procedure, e.g., (!X !Y !Z). Each <e> is either an expression to be evaluated or, if an atomic symbol, a tag which may be transferred to (via GOTO).

(PUSH <var> <exp>)

<var> should be a LISP or FUZZY variable which is bound to a list. The value of <exp> will be CONSed on to the front of the list.

(REMOVE <skel>) or (REMOVE <type> <proc>)

Removes <skel> from the net, failing if <skel> is not present. If the first argument is "GOAL:", "ASSERT:" or "ERASE:", a second argument should be present giving a procedure name, e.g., (REMOVE GOAL: PROC1).

(RESTORE <context>)

<context> should evaluate to a context as returned by SAVE. The net will be restored to its previous state. Default <context> is the state of the net upon entry to the current procedure.

(SAVE <v1> <v2> ...)

Saves the current net and the value of each of the <v>s (may be either LISP (X) or FUZZY (!X) variables). The context which is returned should be saved for later restoration via RESTORE.

(STATE)

Prints out the current state of the associative net.

(SUCCEED <skel> <zval>)

Exits from the current procedure, returning a value of (<skel> . <zval>). If <skel> and <zval> are absent, the instantiated procedure pattern and accumulated Z-value are returned.

(SUCCEED? <skel> <zval>)

Identical to SUCCEED, except the procedure may be restarted if invoked by a FORALL. SUCCEED? may only occur at the top level of a procedure or in an if- or for-statement.

(TRY <list> <skel>)

<list> should instantiate to a list of procedure names. Each will be called in turn with <skel> as its argument until one succeeds, at which time its value will be returned as the value of TRY. If none succeeds, TRY fails.

(VAL <exp>)

Returns the value portion of <exp>. If <exp> is missing, the value of the last FETCH or GOAL statement is returned (the value portion of the latest instantiated procedure pattern or for-pattern may also be retrieved via (VAL) until the first FETCH or GOAL statement is evaluated).

(ZAND THRESH: <thresh> <e1> <e2> ...)

Evaluates each of the <e>s, returning the value of the last <e> evaluated and a Z-value equal to the minimum of all the <e>s. Fails if the Z-value falls below <thresh> or any of the <e>s fail. Default <thresh> is ZLOW.

(ZNOT <exp>)

Returns the value of <exp> with a Z-value of ZHIGH minus (ZVAL <exp>).

(ZOR THRESH: <thresh> <e1> <e2> ...)

Similar to ZAND except returns the maximum of the Z-values.

(ZVAL <exp>)

Similar to VAL, except returns the Z-value portion of <exp>.

PATTERN FUNCTIONS:

(QUOTE <stuff>) or '<stuff>

Serves to stop instantiation just as it stops evaluation in LISP. Thus if !X is bound to A, ('!X !X) instantiates to (!X A).

(*! <name>) or !<name>

Returns the FUZZY value of <name>. If <name> is unbound, acts like ?<name>. May also be used with a skeletal argument to cause instantiation where evaluation would normally occur.

(*!! <name>) or !!<name>

If <name> is bound to a list, it is spliced into the skeleton at this point. If <name> is bound to an atom, "!!" has no effect. If <name> is unbound, acts like ??<name>.

(*& <exp>) or &<exp>

Returns the LISP value of <exp>. May be used to cause evaluation where instantiation would normally occur.

(*&& <exp>) or &&<exp>

If <exp> evaluates to a list, it is spliced into the skeleton at this point. If <exp> evaluates to an atom, "&&" has no effect.

(*?) or ?

Matches any single item.

(*? <name>) or ?<name>

Matches any single item, binding the item to the FUZZY variable <name>.

(*??) or ??

Matches a segment of zero or more items. Subsequent failures by the matcher cause another item to be matched.

(*?? <name>) or ??<name>

Matches a segment of zero or more items, binding the segment matched to the FUZZY variable <name> (<name> is initially given a value of NIL).

(*??/ <name>) or ??/<name>

Matches a segment of zero or more items, binding the length of the segment matched to the FUZZY variable <name>.

(*AND <pat1> <pat2> ...)

Succeeds only if each of the <pat>s match. For example, (*AND ?PAIR (?X ?Y)) will match an ordered pair, assigning the pair to !PAIR and the first and second items to !X and !Y respectively.

(*ANY <pat> <list>)

<list> should instantiate to a list. *ANY will attempt to match <pat> only if the matchee is EQUAL to a member of <list>.

(*CON <pat> <object>) or (*CONTAINS ...)

<object> should instantiate to an arbitrary LISP object. *CON will attempt to match <pat> only if the matchee contains <object> at some level. For example,

```
(FETCH (*CON ? RED))
```

will retrieve the assertion with the highest Z-value which contains RED.

(*INST <pat>) or (*INSTANTIATED ...)

Attempts to match <pat> only if the matchee is fully instantiated.

(*LEN <pat> <n>) or (*LENGTH ...)

Matches <pat> against a segment of length <n>, e.g.,

```
(MATCH [A (*LEN ?X 2) D] [A B C D])
```

binds !X to (B C).

(*NOT <pat> <test-pat>)

Attempts to match <pat> only if <test-pat> does not match. For example, (*NOT ?X (*OR A B)) will match anything except A or B, binding it to ?X.

(*OPT <pat>) or (*OPTIONAL ...)

Indicates that <pat> is optional. If it doesn't match it is ignored.

(*OR <pat1> <pat2> ...)

Succeeds if one of the <pat>s matches.

(*R <pat> <fn1> <fn2> ...) or (*RESTRICT ...)

Applies each <fn> (a LISP function) to the prospective matchee, and then attempts to match <pat> only if none of the <fn>s return *FAIL or NIL.

(*REP <pat> <n>)

Matches <pat> against the next <n> items. Fails if <pat> fails to match an item. For example, (*REP ? 5) will skip the next five items. If <n> is missing, *REP continues until <pat> fails to match, e.g., (*REP X) will span a string of X's.