Tokenized Rule Based System

Steven M. Lewis

Department of Biomedical Engineering
University of Southern California
Los Angeles, CA 90089

Abstract

A rule processing system written in FORTH is described. Clauses may be strings of text words, executable FORTH code or implicitly executable code allowing implicit access to variables and supporting checks that data is known before it is used. Data structures for clauses and rules are described which allow efficient forward or backward chaining. Finally principles of design of expert systems are discussed.

Introduction

Expert systems are finding increasing application in a variety of fields. Park [PAR83] published a simple expert system in FORTH, EXPERT2. In using EXPERT2 in a course, the author became convinced that a better implementation would run faster, generate more readable code and add new functions, specifically the ability to deal with variables and arrays without reverting to FORTH code.

This latter property is important in useable systems because neither the domain experts, who supply knowledge to the system nor the knowledge engineers who encode that knowledge, can be expected to have familiarity with the FORTH language. Because the converse is true—FORTH users cannot be expected to be familiar with the concepts and vocabulary in expert systems. The remainder of the introduction is a short tutorial.

In the purest form, an expert system is a program capable of emulating the performance of a human expert in a specific, tightly defined area of action called a domain. Usually knowledge about the domain is codified in the form of rules. The program, or inference engine, can manipulate the rules and apply them in specific situations. The engine contains no knowledge about the specific problem, merely code for manipulating data and rules. All knowledge of the domain is in the specific rule set. A rule-based system will typically consist of two parts: a rule compiler capable of reading rules and compiling them as an appropriate data structure and a rule interpreter capable of drawing conclusions from the rules and additional information supplied by the user.

A typical rule might look like the example below taken from a set of rules used to classify animals:

```
IF THE ANIMAL IS A MAMMAL
AND IF THE ANIMAL CHEWS CUD
AND IF THE ANIMAL HAS TOED HOOFS
THEN THE ANIMAL IS AN UNGULATE
```

In the above example, each line constitutes a *clause*. The clauses starting with `IF` or `AND IF` are called *antecedents*. The clause starting with `THEN` is a *consequent*. A rule can be fired by the system if all antecedents are true. Firing a rule sets all consequents to true. In the above example the clauses are strings of text having no meaning to the rule interpreter other than a string of characters or words. These will be referred to here as *text clauses* to distinguish them from other types of clauses.

A more sophisticated rule might include other forms of clause including executable words. This concept is discussed in greater detail below.

Rules must be distilled from knowledge of an expert in the field—if not written by the expert, then at least in close consultation with him. In general, it is safe to assume that the expert is probably not a programmer and certainly knows little about FORTH. Even the knowledge engineer who may help an expert to codify knowledge cannot necessarily be expected to be highly familiar with programming, especially in FORTH. With the above in mind, the following objectives can be defined for a rule-based system:

1) The rules must be capable of being read and written by a non-programmer expert. A corollary would state that use of FORTH code should be minimized. The inference engine should provide rich enough data structures to allow most rules to be written in something approaching natural language.

2) The inference engine should be capable of documenting its reasoning. When presenting the user with a conclusion, it should be able to explain the sequence of logic leading it from the user's input to that conclusion. This allows the expert to compare the reasoning of the computer with his own approach and, when conflicts arise, correct the computer.

3) The inference engine must have a means to access FORTH code. While there should not be many rules in code, this allows a computer to gather data from its environment without requiring input from the user. Inputs from instruments, data bases or previously stored responses give real systems a greatly expanded range of capabilities.

The ability of FORTH to define data structures can be used to create a range of structures required to deal with rules in a compact and efficient manner. Most of this paper is really a description of useful data types.

Methods

Rules (Figure 3) are lists of clauses. A rule is compiled when a) the vocabulary **RULES** is current, and b) the words **IF** or **CAVEAT** are encountered. **IF** is the usual rule initiator while **CAVEAT** allows rules without antecedents which can be fired to initialize the system. When a rule is compiling, the words **IF**, **THEN**, and **BECAUSE** will terminate the current clause and start a new one. If the words **IF** or **CAVEAT** are encountered after a consequent starting with **THEN** has been compiled, the current rule is terminated and a new rule started. The word **DONE** will stop all compilation of rules. Thus **IF**, **THEN**, **BECAUSE**, **CAVEAT** and **DONE** are reserved by the rule compiler. The words **ELSE** and **AND** have no special meanings except that the word **AND** may be added before **IF** for readability.

The object of an inference engine is to determine the truth of propositions or clauses. In this section two classes of clauses will be considered: text clauses which are statements in English and must be proven using the logical reasoning processes described below and executable clauses which are executable code and return truth by executing specific code. Executable clauses may be further subdivided into explicitly and implicitly executable clauses. Explicitly executable clauses are simply FORTH code executing to leave a flag on the stack. Implicitly executable clauses are more readable and allow manipulation of numeric data and variables without dropping into FORTH. They also perform more error checking for the user, an important feature.

Consider a sample collection of rules:

```
IF THE ANIMAL HAS FEATHERS
AND IF THE ANIMAL LAYS EGGS
THEN THE ANIMAL IS A BIRD
```

```
IF THE ANIMAL IS A BIRD
AND IF THE ANIMAL SWIMS
THEN THE ANIMAL IS A PENGUIN
BECAUSE PENGUINS ARE THE ONLY FLIGHTLESS SWIMMING BIRD
```

The clauses starting with IF are called antecedents. (The AND at the start of lines has no meaning and is used purely to aid readability.) Clauses starting with THEN are consequents. If all the antecedent clauses in a rule are true, then the inference engine can operate or 'FIRE' and set the consequent clauses to true. Note that the clause "THE ANIMAL IS A BIRD" is a consequent in the first rule and an antecedent in the second. If one wanted to prove that an animal was a bird, they could do so by proving the first rule. In typical systems most clauses will be consequents in one rule and antecedents in others. These will be referred to as *intermediate clauses*.

In the above example the following clauses: THE ANIMAL HAS FEATHERS; THE ANIMAL LAYS EGGS; and THE ANIMAL SWIMS are used only as antecedents. Clauses used only as antecedents are termed *facts*. Because a fact cannot be proven by firing a rule, the only way to determine the truth of a fact is to ask the user.

The clause THE ANIMAL IS A PENGUIN is used only as a consequent. In a logically constructed system such clauses are included because they are end points of a logical chain of reasoning, although they may form the beginning of the reasoning process. Clauses which are consequents but not antecedents are called *hypotheses*. The rule-based system will attempt to acquire data to determine the truth of the hypotheses in the rules.

The clause BECAUSE PENGUINS ARE THE ONLY FLIGHTLESS SWIMMING BIRD is in the rule only for explanation. It is treated like a comment by the system and used only when the system is explaining its reasoning.

It is important that words which manipulate clauses be able to access rapidly the rules which use those clauses. A logical clause data type will be able to access that information. Figure 2 shows the clause data type.

The simplest form of clause is a *text clause*. All words in the clause are in a special vocabulary called TEXT. Words in TEXT with very few exceptions execute as NOOPs. When the rule compiler encounters a word which is not in the text vocabulary, it is simply created. Thus if the rule compiler is presented with the clause:

```
THEN THE ANIMAL IS A TIGER
```

It will compile the words in the TEXT vocabulary. The unknown words will be created as NOOPs and compiled with the clause. This requires that the word creation process have access to the top of the dictionary. Clauses and rules must be assembled in other areas of memory and copied into the dictionary at the end of compilation.

A second type of clause is written directly in FORTH code. The clause:

```
IF RUN A @ B @ * C @ - 100 > ENDRUN
```

is of this form. The word RUN immediately invokes the FORTH interpreter until the word ENDRUN turns it off. RUN also stores the word EXECUTE-ACTIONS (see code) in the action cell for that clause causing execution when the system attempts to prove the clause. Executable clauses can expect no arguments on the stack and must return a single boolean argument.

A variant on the executable clause will have the following form:

```
EVALUE FLOW
EVALUE PRESSURE
IF FLOW IS GREATER THAN ( 100 * PRESSURE )
```

Here FLOW and PRESSURE are declared as expert objects. When invoked they make the clause executable and at run time place their contents on the stack. GREATER is an expert operation using

infix notation for readability which will cause execution of > at the end of the clause. This form of clause is used to enhance readability of clauses for experts who would have difficulty with FORTH code. EOBJECTs are also tested to make sure that the contents have been set to some value. If any object accessed by a clause has not been set, the clause returns false. Objects are set when a consequent of the form:

```
THEN PRESSURE * RESISTANCE -> FLOW
```

sets FLOW to PRESSURE * RESISTANCE. Note that * is in infix notation. A sample rule set, included in the appendix, illustrates these rules.

In the current system, there is little difference at run time between implicitly executable clauses using EVALUEs and explicitly executable clauses using RUN .... ENDRUN with interposed code. Implicitly executable clauses are compiled as FORTH code and at run time are executed. They are, however, easier for inexperienced users to read and write.

Specific Code

The entire code of the system is too long to present here. Instead the inference engine code and a sample rule base will be presented, leaving out the code for clause compilation and the explanatory portions.

The general form of the coding presented here depends heavily on two LISP-like functions: MAPCAR and MAPRULE. MAPCAR takes a FORTH word and applies it to all elements of a list. On execution, the word is passed the address of a list element. If the word BREAK is executed by the operation, further processing is terminated. On exiting, there is a means to test whether BREAK has been used. MAPRULE is similar except the operation is applied to all clauses in a rule. Many words will operate on a rule or a list with BREAK used to show success or irretrievable failure.

The code uses a few words which are not presented for reasons of length, but their use is described in the first section. The system used is Bradley's implementation of F83 under UNIX. This system uses text files instead of screens. The code is presented in linear fashion with no specific line numbers or screen breaks.

Operations

The language SMALLTALK [DUF84] introduced the idea of data objects. An object is a data structure which can only be used for a limited number of operations. The object itself contains the code for the operations which may act on it. In FORTH, objects may be implemented by including within the object a pointer to a list of allowable operations. Clauses are implemented in this manner, allowing the same operators to work on executable and non-executable clauses.

The allowable operations are as follows:

Operations returning a flag

QUERY-CLAUSE: Tell if the item is true, false or unknown. Attempt no operations. TEXT clauses have a truth word which stores this information. Executable clauses execute and return the value.

TEST-CLAUSE: Determine if the item is true, false or unknown based on the current state of the system's knowledge. In this operation backward chaining is permitted but not access to new facts or data. Thus, if a clause is currently unknown, rules which prove that clause may be tested to see if they are currently true. Rules are tested by testing all antecedent clauses to see if they are true. This process can recurse until one of the following happens:

> The system encounters a FACT, a clause which cannot be proven without querying the operator. These queries are not permitted in test mode.

Some rule proving the clause is tested and found true.

All rules proving the clause are tested and cannot be proven.

Executable clauses are tested by execution.
`PROVE-CLAUSE`: Proof returns true or false depending on whether the clause is provable with all knowledge available to the system and its operator. Proof is similar to testing except that, when facts are encountered when attempting to prove a rule or clause, the operator will be queried as to the truth or falsehood of the fact. In the proof operation, unprovable and false are identical.

Operations with no return argument
`SET-TRUE`: The truth word is set to true. In this operation, no attempt is made to examine the implications of the change.
`SET-FALSE`: Set the truth word to false. Note that false and unknown are not identical in expert systems. False indicates that the system has data proving the proposition false.
`MAKE-TRUE`: Set the truth word true. Then test all rules using the clause as an antecedent. If the rule becomes true, then apply the `MAKE-TRUE` operator to all consequents. Continue until no further operations are allowed.
`PRINT-CLAUSE`: Print the clause. Printing clauses consists of decompiling the clause. Text clauses are exactly reproduced by decompilation since in the `TEXT` vocabulary the words are simple dictionary entries. Decompilation of `RUN ... ENDRUN` will produce the generating FORTH code except for ambiguities about branch instructions (`IF, THEN, ELSE, BEGIN, WHILE, UNTIL`). Decompilation of implicit clauses generates the underlying FORTH code rather than the generating text.
Thus:

```
IF PRESSURE IS GREATER THAN FLOW
```

decompiles as

```
IF 34089 @ IS THAN 35673 @ >
```

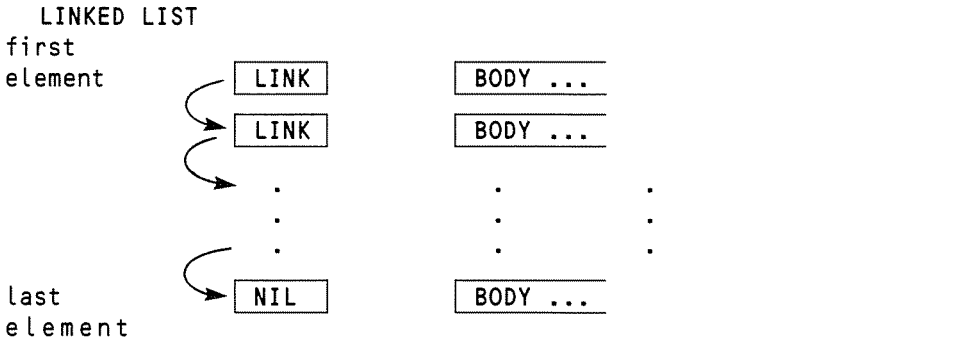
since `PRESSURE, FLOW` and `GREATER` are immediate words compiling the desired operations.

The operations `TEST` and `PROVE` perform backward chaining, proving a clause by examining rules which prove the clause and continuing backwards. `MAKE-TRUE` allows forward chaining, carrying the implications of a fact to rules using that fact.

Data Structures

A basic structure used throughout the system is the linked list (Figure 1). An unhashed FORTH dictionary is a good example of a linked list with each element containing a pointer to the next until a terminating condition is reached. This implementation is similar to the FORTH dictionary in that the object following the link is of unspecified size. Unlike LISP, operators on lists are necessarily specialized to the specific structure which follows the link cell.

All clauses (Figure 2) are linked together in a linked list. Each clause has in its header a cell indicating its truth. Currently only fixed values indicating known-true, known-false and unknown are used but the same system can be used to estimate levels of certainty. Linked lists of rules which use the clause as an antecedent and which prove the clause are included to allow for rapid forward and backward chaining. When the system wishes to operate on the clause, it fetches the desired operation from the structure pointed to by the `ACTION` cell. The body of the clause is identical with the body of a FORTH word terminating in a `NEXT` operation. Executable clauses merely place the address of the clause body on the return stack and perform a `NEXT`, which executes the body of the clause. Execution should leave a single boolean flag on the stack.



EXAMPLE of a LINK: RULE LIST used by clauses

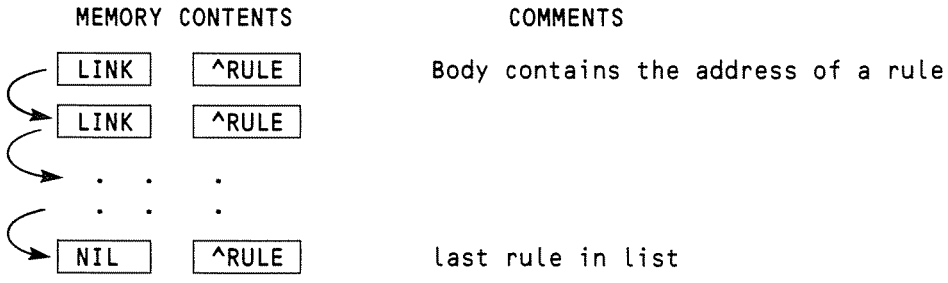


Figure 1

CLAUSE STRUCTURE

FIELD		COMMENTS
Link	LINK	Next clause in clause list
Truth	TRUTH	0 unknown, 100 true
Used-by	^RULE LIST	If nonzero points to list of rules where cause is antecedent
Set-by	^RULE LIST	If nonzero points to list of rules where clause is consequent
Special	^EVALUE LIST	If nonzero points to a list of EVALUES use in clause
Action	^ACTION	Points to action structure

Body	'word	first word in body
.		
.		
.		
	(;)	Do NEXT to leave

Figure 2

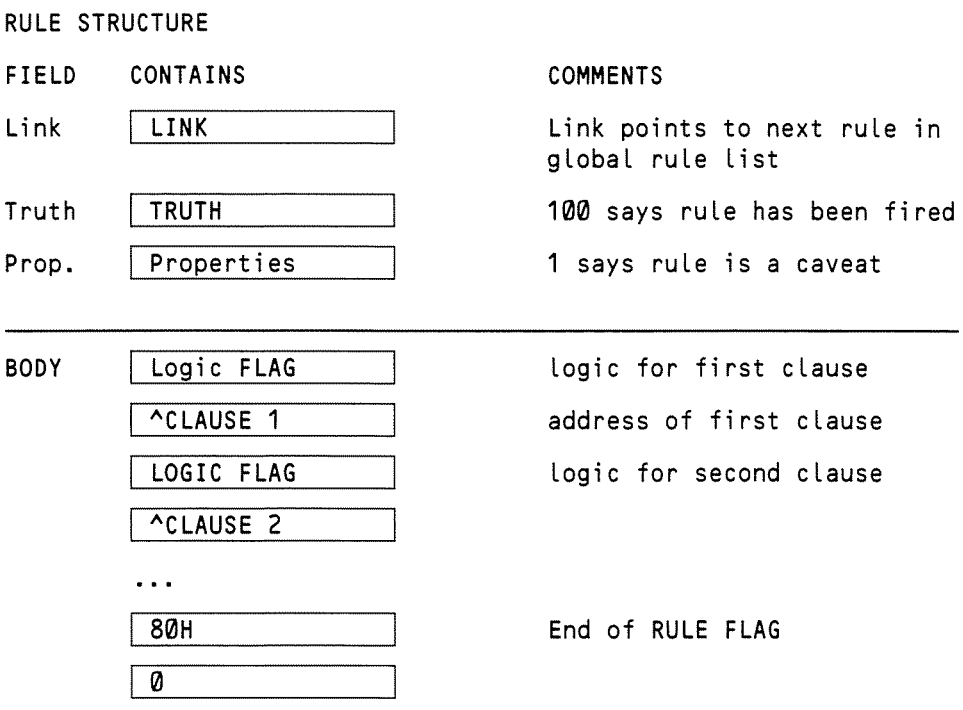


Figure 3

Rules (Figure 3) are treated as lists of clauses together with flags describing the use of the clause. The logic flag is one cell with bits indicating appropriate information. Rules consist of a link field, a truth field which tells whether the rule has fired and a list of logic flags followed by clause addresses. All rules are linked in a list which allows for initialization or sequential testing of all rules.

Logic Flag Bits

0	NOT	reverse the flag returned by the clause
1	IF	clause is antecedent
2	THEN	clause is a consequent
3	BECAUSE	clause is explanatory
4	RUN	clause is executable
5,6		not used
7	LAST-CLAUSE	terminates a rule

Inference Engine

The inference engine takes the rules and attempts to prove clauses from them. There are three general schemes for inference engines. A backward chaining system will start with a hypothesis and work backwards through the tree of rules and clauses proving the hypothesis. A forward chaining engine will start with facts and prove rules until one or more hypotheses are proven. A looping engine will test all rules and continue to fire rules until explicitly stopped. In this later case, the facts will have to change or the looping is relatively pointless.

Backward Chaining

The word for backward chaining, `DIAGNOSE`, tests each clause in the clause list to see if it is a hypothesis (that is a consequent which is not used as an antecedent). It attempts to prove each

hypothesis by testing each rule in the list of rules which prove that clause (see clause structure above). A rule is proven by testing each antecedent clause. Executable clauses are executed and return a flag. Facts, clauses which are used as antecedents but not consequents, can only be tested by asking the user. These are immediately asked. Intermediate clauses are consequents of other rules. These clauses are tested by attempting to prove a rule which uses them as a consequent. The process of proof proceeded recursively until only executable clauses and facts are encountered. Attempts to prove a rule continue until either all antecedents are satisfied, in which case the rule is proven, or one antecedent cannot be satisfied, in which case the proof of that rule is abandoned and the rule is considered false. Attempts to prove a clause proceed until either a rule fires which proves that clause or all rules which might prove the clause are unsuccessfully tested. In the later case the clause returns false.

If a single hypothesis is proven, the system tests the flag `MUTUALLY-EXCLUSIVE` to see whether to test additional hypotheses or to stop and state the single conclusion.

Forward Chaining

In forward chaining, the system searches the clause list for facts. Facts are queried. Following this, all rules which use the clause are tested to see if all antecedents are true. If so, the rule fires, and sets consequents and the rules depending on how the consequents are set. In the code shown below the process continues until some hypothesis has been proven.

Repeated Proof

In repeated proof mode, the system repeatedly searches the list of rules for rules which are capable of firing. Several systems for ordering which rules are fired are possible, but currently the system simply searches the rules in order. In most examples considered thus far, either only one rule can fire at a time or the order is irrelevant. Two conditions can halt the process. Either the system finds that no rules are capable of firing or a consequent with the single word `STOP` is executed. The code for this mode is not shown.

Repeated proof allows the system to act like a state machine. This mode has been used to construct benchmarks such as the Towers of Hanoi disk moving problem. Serious use of this mode requires some thought about ordering the firing of candidate rules.

Discussion

The project began as a learning experience. The author and a student took an existing expert system, EXPERT2 [PAR83] and attempted to implement an automatic registration system in an undergraduate program in bioengineering. While the task could be performed with that inference engine, virtually all clauses had to be written in FORTH code. The problem was that there was no way to access a structure such as a table of prerequisite clauses and corequisite classes. Without this, a separate rule had to be written for each class. Furthermore, to incorporate the assumption that if a senior class had been taken the prerequisites for it had been fulfilled, required a completely separate set of rules pointing backwards.

Three problems were examined: the PUFF pulmonary function system, a system for pulmonary diagnosis based on the results of a specific class of pulmonary function measurements; a rule-based solution for the Towers of Hanoi problem; and the registration problem. In all cases, simple text clauses were inadequate for most rules. In PUFF, virtually all of the rules referred to measured quantities. In Towers, the rules referred to the contents of poles, most usefully considered as stacks. In the registration problem, text- based rules were adequate but highly inefficient for the reasons outlined above; most procedures naturally refer to tables of prerequisites and corequisites.

One solution would have been to descend to FORTH level in most clauses. While this is efficient, it makes the rules unreadable for a non-FORTH programmer. Ideally the rules should be readable and writable by a domain expert and able to be critiqued by experts. This suggests a strong advantage for being able to write rules in at least pseudo-English. This work and others ([JOH83], [PAR83]) suggest that FORTH has the tools to support readable rules and this advantage ought not to be lost.

A second consideration with EXPERT2 is that the inference engine is inefficient. Clauses prove themselves by searching for relevant rules at run time. Since the information is present at compile time, it was felt that it is more efficient for clauses to keep track of relevant rules as they are compiled.

A third consideration is that the clauses are stored as text strings, thus ignoring the advantages of the FORTH compiler, both in storage and in simplification of expressions. Thus in EXPERT2 clauses begin with words like `IFNOT IFNOTRUN ANDIF` etc. Allowing `NOT` and `RUN` to be immediate and ignoring trailing `AND`s improves readability.

Both systems deliberately leave out several possible features. `ELSE` is left off keeping with the logic that false is unknown and only proven rules can act. Equivalent effects may be accomplished with an added rule. `OR` is also left out. In the rules all antecedents are `AND`ed. Defining multiple rules can easily allow the same meaning as `OR`ing several clauses together. `AND`ing all `AND` clauses in a rule to form an intermediate conclusion and then `AND`ing the intermediate with each clause in a series of separate rules is a reasonable way to emulate an `OR`.

An additional property of both this system and EXPERT2 is that rules are tested in the order they are compiled. In situations where more than one rule may be invoked, this property may influence the behavior of the system. While no systems have been examined where order of firing is a problem, clearly in more complex systems with hundreds or thousands of rules this may be an important consideration.

In backward chaining systems, the use of `THEN ... NOT ...` to set a clause to false is discouraged. Clauses are assumed false if they cannot be set by any rule. By eliminating the use of `THEN ... NOT` clauses to falsify a clause, one reduces the possibility of contradictory rules, with one rule setting a clause false and another setting it true. In some situations, it is useful to allow clauses to be falsified. In the Towers game, there are rules which conclude that a disk cannot be moved. After the obstructing disks have been removed, it is necessary to negate that conclusion. Similarly in systems which run in real time, it may be necessary to alter an early conclusion as the system changes.

Conclusions

One of the most interesting conclusions of working with this system is that because of the richness of data structures available in FORTH, complex systems may run on smaller machines and run faster than equivalent systems written in LISP. Workers at General Electric came to a similar conclusion in developing a system for diagnosing problems in Diesel locomotives [JOH83]. In the current system, lists are used extensively for storing permanent data about structures and their relationships. Stacks are much more useful for storing temporary data used while running the inference engine. When lists are used only for permanent storage with items added but never deleted from lists, the need for extensive memory management and garbage collection is reduced.

Of course, there are disadvantages. It has been said of FORTH that you can do anything in the language, but you must write it yourself. The complex data structures are not present in FORTH; they must be added by the user. The system described here is an attempt to show the power of FORTH in dealing with expert systems. Much added work is needed to make FORTH a true AI language capable of challenging more conventional languages such as LISP and PROLOG.

References

[JOH83] Johnson, Harold E. and Piero P. Bonissone. "Expert System for Diesel Electric Locomotive Repair." *Journal of Forth Application and Research* 1 (1983): 7-16.

[PAR83] Park, Jack. EXPERT-2: A Knowledge Based System. Mountain View, CA: Mountain View Press, 1983.

[DUF84] Duff, Charles B. and Norman D. Iverson. "FORTH meets SMALLTALK." *Journal of Forth Application and Research* 2 (1984): 7-26.

Appendix

Used But Not Listed

```
push ( n,stack--<> push a number onto a private stack )
pop  ( stack -- n  pop the top element off a private stack )
ask  ( clause -- f ask the operator if the clause is true.
       If the answer is w for why, dump logical reasoning )
printclause ( clause--<> decompile the clause and print it.
              This handles literals properly )
initialize-inference-engine ( <>--<> housekeeping )
```

These Words Allow the Code to be Written Independent of Stack Width

```
\ CELL is the stack width in bytes
2 constant cell             cell negate constant -cell
: cell* 2*;                 : cell/  2/ ;

: t/f if true else false then ;      \ make a number into a boolean
```

Notation

```
\ ^ clause refers to a pointer to a clause, an address containing
\    the address of the object
\ rules and clauses are passed as the address of the link field
```

Mapping Words—Words to Apply a Forth Word to an Entire Data Structure

```
variable <done>       \ flag to break out of map
: break <done> on ;   \ set the flag
\ save the fact that a BREAK was used in a variable
: save-break ( addr--<> ) <done> @ <done> off swap ! ;
```

Return Stack Access Words

```
: 2r@ ( <>--n return next to top element of return stack )
    r> r> r>   dup >r   swap >r   swap >r ;

\ code version of return stack access word for 68000
code 2r@ 4 rp d) sp -) move c;
```

List Access Words

```
variable nil      \ a dummy address to terminate lists

\ mapcar works like the lisp mapcar applying an operator
\ to each element of a linked list.

variable map-broken    \ stores whether last map was broken
```

```
: mapcar ( list,operation--<> operate on each element of list)
    <done> off ( initialize <done> )
    over  if   ( make sure this is a list )
       >r >r ( put operations on return stack )
       begin
        <done> @  r@ nil =  or 0=  while ( done conditions? )
        r@ 2r@ execute  ( operate)  r>  @ >r  ( next element)
       repeat r> r>
    then 2drop
    map-broken save-break ;  ( remember if break invoked )

\ Map an operation to all clauses of a rule

variable rule-broken  ( flag used by map-rule for break)
\ map-rule apply an operation to a rule
\ NOTE >>
\ The words next-clause and first-if+ are defined below
\ in the rules section map-rule is listed here for continuity
: map-rule ( rule,operation--<> )
    <done> off
    >r first-if+ >r
    begin (
        <done> @ r@ cell- @ rule-end and or not while (done conditions)
        r@ 2r@ execute                \ do operation
          r> next-clause >r           \ go to next clause
     repeat r> r> 2drop               \ clean up
     rule-broken save-break ; \ save break information
```

Pointers to Rule and Clause Lists

```
variable ^clauses     \ a pointer to a list of clauses
variable ^rules       \ a pointer to a list of rules
: first-clause ^clauses @ ;     \ address of the first clause
: first-rule   ^rules   @ ;     \ address of first rule

\ offset words this creates words to take an address and
\ add an offset
: do-offset
    create ,     ( offset-- <> )
    does>         ( addr -- addr+offset )
        @ + ;

\ offsets for dealing with rules
cell constant lf-size              \ size of logic flag
  cell  do-offset truth+          \ move to truth field
2 cell* do-offset properties+    \ properties field 0=rule 1 caveat
3 cell* lf-size + do-offset first-if+ \ move to first clause pointer
lf-size cell + do-offset next-clause  \ move from one clause to next
```

```
\ truth and falsehood tests identical for rules and clauses
100 constant truth   \ this allows later versions to have scales
                     \ of truth from 0=unknown to 100 = surely true
-100 constant falsehood  \ in certain cases we want to mark a clause
                         \ false
: true? ( rule -- f returns t if rule fired )
    truth+ @ truth = ;    \ NOTE this also works for clauses
: false? ( rule -- f if rule is known false
    truth+ @ falsehood = ;

\ clause offsets
2 cell* do-offset use+
3 cell* do-offset set+
4 cell* do-offset special+
5 cell* do-offset action+

\ specific clause tests all take clause, return flag
: fact? dup use+ @ 0=    \ not consequent
        set+ @ 0= not and \ and if antecedent
: hypothesis? dup set+ @ 0=  \ is consequent
        use+ @ 0= not and  \ and not antecedent

\ action offsets and actions
: do-action ( create> offset-to-action  does>clause -- ?? )
    create ,   ( offset -- offset to action )
    does> ( clause--?? return depends on action )
    @ over action+ @    \ get offset, action field addr
    + @ execute ;                   \ and do the requested action
\ now create words to perform specific actions
0      do-action prove-clause    \ prove the clause
1 cell* do-action query-clause   \ test truth of clause
2 cell* do-action print-clause   \ print the clause
3 cell* do-action set-true       \ set the clause true
4 cell* do-action set-false      \ set the clause false
5 cell* do-action make-true      \ set true and then forward chain
6 cell* do-action test-clause    \ test truth and to see if proveable

\ tests of logic flag
: testlogic      \ defining word for tests of logic flag
    create ,   ( bit to test )
    does>     ( ^clause -- ^clause,f true if bit is on )
        @    \ get the bit to test
           over cell-  @ \ get the logic flag in the previous cell
        and t/f ;    \ compare and make boolian

\ easy tests of clause logic all take clause in rule
\ clause-addr--clauseaddr,logic
1   testlogig is-not?            \ negative logic
2   testlogic is-if?             \ antecedent
4   testlogic is-then?           \ consequent
8   testlogic is-cause?          \ because clause
16  testlogic is-run?            \ executable
128 testlogic is-end?            \ rule end
```

Backward Chaining Section of the Engine

```
: do-proof ( clause--f )
    dup subhyps push                \ save for why on private stack
    dup query-clause if drop true   \ already done
          else prove-clause then    \ else do clause action
    subhyps pop drop ;

: prove-antecedent ( ^clause--<> break if antecedent false )
    is-not? swap                    \ remember negative logic
    is-if? if @ do-proof            \ if antecedent prove
           xor                      \ if NOT this reverses logic
              not if break then     \ false then break
          else 2drop then ;

\ PROVE RULE
: prove-antecedents ( rule--f )
    ['] prove-antecedent map-rule   \ prove all antecedents
    rule-broken @ 0= ;                 \ break shows failure

: set-consequent ( ^clause--<> set consequent )
    is-then? if @ set-true then ;

: set-consequents ( rule--<> )
    ['] set-consequent map-rule ;

: prove-rule ( rule--f )
    dup rulestack push              \  save on private stack for why
    dup prove-antecedents           \ try to prove all antecedents
    if dup set-truth                \ remember the rule has fired
        set-consequents true        \ and set the consequents
     else drop false then
    rulestack pop drop  ;           \ clean up the trace stack

\ CAVEATS
\ caveats are rules without antecedents and may be fired at
\ the beginning of operation to perform initialization
\ they are marked with a 1 in the rule's property field
1 constant caveat
: caveat? ( rule -- f )
    properties+ @ caveat = ;
: do-caveat ( rule--<> prove caveats )
    dup caveat? if prove-rule then drop ;
\ execute all caveats
: do-caveats first-rule ['] do-caveat mapcar ;

\ PROVE HYPOTHESES the basis of a backward chaining engine

\ NOTE if mutually-exclusive is true we prove at most one hypothesis
\ if not all are tested
: prove-hypothesis ( clause--<> if hypothesis prove )
    dup hypothesis? if prove-clause         \ return flag
                    if mutually-exclusive @  \ if mutually exclusive set
                        if break then then \ then break to quit
                else drop then ;
```

```
: prove-hypotheses ( <>--<> )
    first-clause ['] prove-hypothesis mapcar ;

: print-conclusion ( clause--<> )
    dup hypothesis?                    \ only consider hypotheses
        if dup test-truth              \ if the clause is true
            if print-clause            \ then print it
                something-proven on    \ remember we proved something
            else drop then
        else drop then ;


\ THIS WORD IS THE BACKWARDS CHAINING ENGINE
: diagnose
    init-inference-engine          \ housekeeping not listed
    do-caveats                     \ execute caveats if any
    prove-hypotheses               \ prove 1 or more hypotheses
    something-proven off           \ flag to see if anything proven
    first-clause ['] print-conclusion mapcar \ print conclusions
    something-proven @ not         \ message if nothing proven
      if cr .'' cannot prove anythng '' then ;
```

Forward Chaining

```
: test-antecedent ( ^clause--<> break on false )
    is-not? swap                       \ remember negative logic
    is-if? if @ test-clause            \ test to see if clause true
              xor not if break then    \ break if antecedent not true
           else 2drop then ;

: test-antecedents ( rule -- f true if all antecedents known true )
    ['] test-antecedent map-rule       \ test all antecedents
    rule-broken @ 0= ;          \ true of no breaks all clauses true

: test-rule ( rule -- <> fire rule if all clauses true )
    dup true? not                      \ if the rule has not fired
    if dup test-antecedents            \ are all antecedents true
        if dup chain-consequents       \ carry implications forward
           set-truth                   \ remember rule has fired
        else drop then
    else drop then ;

: chain-consequent ( ^clause -- <> forward chain consequents )
    is-then?                           \ consequent?
    if @ make-true then ;              \ chain forward with set

: (chain-consequents) ( rule--<>  chain all consequents forward )
    ['] chain-consequent map-rule ;
\ now resolve forward reference
' (chain-consequents) is chain-consequents
```

```
: hypothesis-proven? ( clause -- <> break if clause is true
 hypothesis )
    dup hypothesis?
    if    true?
        if break then
    else drop then ;

: break-on-proof ( <>--<> if any hypothesis is proven break )
    first-clause ['] hypothesis-proven? mapcar \ test all clauses
    mapbroken @                          \ any hypotheses proven
    if break then ;                      \ if so break

: forward-fact ( clause -- <> ask facts and forward chain )
    dup fact?
    if dup ask                           \ this returns true or false
        if        \ only true facts count in this system
            set-clause     \ set clause true and all implications
            mutually-exclusive @      \ if only one hypothesis true
            if break-on-proof then    \ break if anything proven
        else drop then
    else drop then ;

\ THIS WORD IS THE COMPLETE FORWARD CHAINING ENGINE
: forward-chain ( <> -- <> complete forward chaining system )
    initialize-inference-engine
    do-caveats
    first-clause ['] forward-fact mapcar \ forward on all facts
    print-conclusions ;
```

Action Vectors

```
\ TEXT CLAUSES
: set-truth ( clause--<> ) truth swap truth+ ! ;

: set-falsehood ( clause--<> ) falsehood swap truth+ ! ;

: set-forward ( clause--<> set truth and chain forward )
    dup set-truth
    use+ @                      \ get pointer to affected rule list
    ?dup if                     \ if any
        ['] test-rule mapcar    \ test all of them
                                   then ;

: test-rule-break ( rule -- <> break if antecedents all true )
    fetch dup test-antecedents\ get rule and test
    if dup chain-consequents  \ set consequents true
        set-truth break       \ remember rule fired then break
    else drop then ;
```

```
: do-query ( clause -- f return if known facts prove the clause )
    dup test-true                          \ is the clause known true
    if drop true                           \ yes then done
    else dup set+ fetch                    \ is clause a consequent?
        ?dup if
        fetch ['] test-rule-break mapcar \ any rule ready
            mapbroken fetch                \ break means true
        else drop false then then          \ not consequent then leave


: prove-rule-break ( rule --<> generate break if rule proveable )
    prove-rule if break then ;

: verify ( clause -- f backward chaining proof )
    dup true?             \ is the clause alread known to be true
    if drop true          \ then report so
    else dup fact?        \ is this a fact? if so no proof possible
        if dup ask        \ so ask the operator
            if set-true true            \ remember clause is true
            else set-false false then   \ remember clause is false
        else
            set+ @        \ get rules which can prove this clause
            ['] prove-rule-break mapcar \ prove rule break if
                                        \ proven
            mapbroken @   \ break shows proof else not provable
    then then ;

\ MAKE TEXT CLAUSE ACTIONS

variable text-actions -cell allot  \ create header
' verify          ,    \ verify proves the clause
' true?          ,    \ test truth without proof
' printclause    ,    \ print the clause by decompilation
' set-truth      ,    \ set clause true
' set-falsehood  ,    \ set clause false
' set-forward    ,    \ set true and chain forward
' do-query       ,    \ query-clause does this

\ EXECUTABLE CLAUSE ACTION VECTORS

: do-run ( clause -- <> execute body of the clause )
    body+ >r ;      \ point the return stack at the body and do semis
                    \ the body always ends with the word compiled by ;
                    \ in most systems this will properly execute and
                    \ return

variable executable-actions -cell allot  \ create header
' do-run         ,    \ execute to prove
' do-run         ,    \ execute to set
' printclause    ,    \ print by decompilation
' drop           ,    \ never remember truth always execute
' drop           ,    \ the same for falsehood
' drop           ,    \ no forward chaining
' do-run         ,    \ execute for truth query
```

A Sample Rule Set

This set of rules diagnoses simple problems in an artificial kidney. It is included to demonstrate some of the arithmetic capabilities of the rule processing system. Parterial is the inlet blood pressure, Pven is the outlet blood pressure and resistance is the resistance of the artificial kidney. GETPART reads the arterial pressure and leaves the answer on the stack. GETVEN and GETFLOW are equivalent operations for venous pressure and flow respectively.

```
TEXT DEFINITIONS

EVALUE Parterial    \ declare arterial pressure
EVALUE Pven                  \ venous pressure
EVALUE FLOW         \ flow
EVALUE RESISTANCE   \ resistance

RULES               \ start compiling rules

\ This 'rule' reads in values for pressure and flows.  As a CAVEAT it
\ has the following properties; 1) the rule starts without an If and may
\ have no if statement. 2) In most systems CAVEATS are executed once at
\ the start of rule processing and are thus useful for initialization of
\ variables .

CAVEAT
Then GETPART -> Parterial
Then GETPVEN -> Pven
Then GETFLOW -> Flow
Then ( Parterial - Pven ) / FLOW -> RESISTANCE


If RESISTANCE is greater than 2000
Then the kidney is clogged
Then Alarm

If Parterial is less than 50
and if flow is less than 10
then the arterial line is disconnected
then Alarm

If Pven is greater than 40
and if RESISTANCE is less than 2000
then the venous line is obstructed

If the kidney is not clogged
and if the arterial line is not disconnected
and if the venous line is not obstructed
then the system is normal
```