ARTIFICIAL INTELLIGENCE PACKAGE EXPERT SYSTEM TOOLBOX

for SUPER-FORTH 64™

By Jack Park

Available at the Vintage Volts website

http://www.vintagevolts.com

EXPERT-2
A Knowledge-Based Consequent Reasoning Computer Program Toolkit

CONTENTS

Acknowledgements

EXPERT-2 is the original work of employees of NIMBLE, a company located in Brownsville, California. Specifically, Jack Park created the original EXPERT-1, with diagnostic and compiler contributions of Dan Wood.

EXPERT-2 is the result of the lessons learned from EXPERT-1, and contribution of valuable FORTH programming techniques from John Cassady, of Oakland, California.

CHAPTER 1

Introduction

Within the field to artificial intelligence, one particular activity has recently emerged as immediately applicable to a variety of users, and transferable from the laboratory to the field. This activity focuses on the creation, and use of so-called "expert" or "knowledge-based" computer systems.

Expert computer systems have the potential power to make available human knowledge and expertise to a vast array of users. The possibility of replicating concentrated human expertise has spawned an emerging industry in knowledge engineering - a field devoted to translating human knowledge into expert computer programs.

This document presents a brief overview of expert systems and knowledge engineering, and details the use of a small, but powerful inference machine, a program which allows you to create and run expert programs. EXPERT-2, the inference program presented here, is written in MVP-FORTH. This document gives the source listing of EXPERT-2 in the Appendix.

The MVP-FORTH Environment

EXPERT-2 is written in MVP-FORTH, one of several "standard" FORTH dialects. EXPERT-2 is written in such a manner that any of the ambiguities that may arise out of use of this dialect are reduced, or are obvious, and will be caught by the compiler.

EXPERT-2

EXPERT-2 is intended to be a tool-kit for experimenters and developers of expert computer systems. EXPERT-2 includes facilities for compling expert rule programs, and for performing logic inferences - reasoning - on those rules. Notes are included in the listing and in this document for possible changes and extensions to EXPERT-2 to create even more powerful expert inference machines.

This Document

This document is intended to serve as an overview of expert systems, and as an introduction to EXPERT-2. This document is by no means intended to be a state-of-the-art dissertation on the field of knowledge engineering. Several good books are now available on that subject, and many periodicals detail the development of various expert systems. A brief bibliography is included to provide readers with some ideas on where to go for more information.

CHAPTER 2

Expert Systems Overview

You are an entrepreneur, just starting your own business. Maybe you are a farmer keeping track of your crops in the field. Or, perhaps you are an investor - or speculator - tracking your investments. What do each of these individuals have in common? They all need and use information; they all make decisions based on that information. They may use the services of a consultant to help in their decision process. Such a consultant might be a knowledge-based expert computer system.

The mere idea of using a computer as a consultant is at once courageous, lofty, and somewhat unthinkable. However, after many years of artificial intelligence research, so-called expert systems are emerging as useful, and popular.

Expert computer systems were first proposed in 1943, but it has required recent developments in computer hardware technology and artifical intelligence programming to bring expert programs to the operational, rather than total experimental phase.

Artificial intelligence - as a computer science - has worked to model the human mind in an effort to understand how humans think, how they learn, and how computers might assist in these tasks. Expert computer programs take a small slice of the artifical intelligence pie - that of mimicking how humans reason to make certain types of decisions - and provide this limited reasoning power to the program user.

Decision making by reasoning requires the ability to recognize and sort patterns and events which are useful to the objective at the moment. There seems to be mounting evidence that many of the processes of reasoning, including insight, invention, and inspiration of the reasoner are largely the outcome of a vast amount of mental flounderings, and countless trial and error attempts. Therefore, an expert computer program designed to follow a similar trial and error process, looking for a recognizable pattern without the countless distractions which affect the thoughts of a human appears to offer the possibility of assisting the human reasoning process.

Information used for reasoning is comprised of data - raw, or otherwise preprocessed data. Information is also based on rules for using that data for making decisions, and methods for using the rules. By designing a computer reasoning program to use these three types of information, a knowledge-based expert program is created.

Here are some examples of the raw or preprocessed data used by an expert. The entrepreneur knows how much cash he has stashed away in the bank with which to form his enterprise. The farmer knows which crops he has planted, where those crops are in their growth cycle, what the costs are

for performing certain operations like watering, spraying insecticides, harvesting, and the like. The speculator knows which investments he owns, what the market is doing - perhaps more accurately, has just done - with those types of investment instruments, and what his cash position is in case he chooses one of several optional actions. These, and other bits of information form the data used in a decision making process.

The entrepreneur makes decisions based on his data, and with the help of certain "rules" or knowledge found in various books or through conversations with "experts" in the field of starting small business. These rules are the step-by-step instructions used in starting a business, and various guidelines used in the decision-making process. What to expect in the way of incorporation costs, how much cash to keep in the bank for crisis management, how to apply for a loan at the bank, how to sell products, and other guidelines, in part, form a portion of the data base. Decisions must be made. To look more closely at the entrepreneurial problem, we observe that decisions must be made down to the level of what to do with each available penny. These decisions affect how the entrepreneur sets-up his accounting books, which business costs are entered into certain categories in those books - and which are not, and more. These decisions affect the tax position the company has when it achieves a profit, the amount of profit it has for investor's dividends, for research, and for expansion. Rules might be devised which help the entrepreneur maximize his profits, or cash flow, for example. These rules are either learned through experience, gleaned from books, or gained through the expertise of consultants.

The farmer and the speculator each use various rules to guide their operations. Such rules might help clarify a problem at hand, list procedures to use to solve a problem or prove a theorem, list assertions which may affect the problem, or test judgmental information for reliability. These rules, while not necessarily the same rules used by the entrepreneur, perform the same function of guiding decision making, and are available from the same source - books, experience (sometimes called "boot-strap rules"), or consulting experts.

Many of the rules we use in our daily business activity can be written into computer programs which help us conduct our affairs. Some rules, however, cannot easily be coded. These rules are the intuitions, hunches, or wild guesses we use - like coin flipping, or executive decision-maker dart boards. Rules are needed, then, to guide the decision making process. At some point in that process, it may become appropriate to use the familiar random number generation routine to guide those decisions left to wild guesses. Hunches could be written as rules, but those resulting rules risk inheriting the qualities of proven facts, and might lose the sense of adventure found in hunches.

The Rule Base

Rules can be created which guide the decision making process. Just how are these rules written? Two different ways to create rules for a computer program may be considered:

- Writing rules directly into a program.
- Writing rules outside the program, as a rule or expert knowledge base.

By using IF-THEN statements within any program, you are imbedding rules in that program. As the program is being executed, the computer eventually runs into one of your IF statements, performs the test indicated by the program code, and branches to another program routine according to the results of the test. Such a program could use as one of its tests a question to the operator. The operator's answer forms the actual test performed. The question could be stated to require a TRUE or FALSE response which would directly guide the consequent THEN action of your IF-THEN test. Or, the question could ask for a new bit of information, like, how much cash is left in the petty cash jar. The response, a dollar value, could be compared to another value say, the minimum cash level required in that jar and the truth of the rule determined; true - the amount available is greater than the minimum, or false - the amount is less than minimum. The rule will then direct the program to move on to its next action. If the program is written to instruct the operator to transfer more cash into the jar in the event the amount fell below minimums, then you might conclude the computer is simply acting like a transistorized office manager. But, suppose that another rule had been written which triggers an executive alarm any time the petty cash jar has less than a certain amount of cash. That rule might motivate a decision to put a lock on the jar, or worse.

Another way to code rules is to create a "rule language". This rule language would be another high level language like BASIC, PASCAL, or MVP-FORTH. Any high level language is created to allow the computer user to write programs using statements that are easily read by humans, but which can also be understood, with the help of an interpreter or compiler, by the computer. An expert rule interpreter (also called an inference interpreter, or inference machine) can understand and perform operations on a list of rules. The difference between these expert rules used by a rule interpreter, and those imbedded inside an ordinary program, is that the expert rules are separate from the rule interpreter. These rules form the "expert program" and can be passed around, read, changed, or otherwise improved upon without any effect on the rule interpreter itself. Because of this, expert humans can create rule bases, just like individuals now create programs in BASIC or other language. They need not worry about how the rule interpreter goes about its process of deducing answers, testing

results, or offering comments on why it concluded something. They only need to be concerned with the art of translating human expertise into an appropriately-coded set of rules.

By imbedding rules into the program, one needs to be concerned with the entire program. By coding the rules as if they were another high-level language, one need not be involved with the intricacies of the rule interpreter program. This separation of rules and rule interpreter offers several advantages:

- o The rule interpreter can be used for a variety of applications.
- o Different experts can be called-up to create rules without being involved with the interpreter.
- o Rules can be developed and tested incrementally.

With a sufficiently general rule interpreter, rule bases can be developed for guiding farm operations, helping-out with office operations, setting-up new business, and, perhaps, assisting in investments. To be sufficiently general, the rule interpreter must allow for two different rule types: those rules concerned with concepts, ideas, or assertions, and those concerned with data manipulation. Some rules must determine the truth value of "aphids were seen today for the first time" and others must determine if the Dow-Jones averages had "over a 10% drop in price at the same time as a greater than 25% increase in trading volume." The first example could rely on some sort of electronic aphid sensor out in the tomato patch, but since no such sensor exists, it's simpler - at this stage of the technology - to ask the operator if the statement "aphids were seen today for the first time" is true. The operator checks the field reports, or hollers out the window. Maybe the computer asking the question is along for the ride in a tractor, and asking the operator is the same as asking the person charged with the responsibility with checking the aphid pheromone traps.

Notice the two distinct types of data used above to illustrate how a computer program might represent information it needs; it could ask for a TRUE or FALSE representation of whether the Dow Jones industrial averages have achieved certain conditions, or it could ask for those values necessary and calculate, from a set of equations and rules, if those conditions have been met. The first representation is a "qualitative" one, while the second is "quantitative". Qualitative information represents the scene in which the program operates in terms of general features, and relative values. Quantitative information represents the same scene using mathematical equations that manipulate specific numerical values. The question above about stock price and volume could just as easily be treated as a concept or assertion - a qualitative bit of knowledge - and asked of the operator, but then the operator would be observed fiddling with a pocket

calculator, a stack of newspapers, and pencil and paper. That's hardly what the expert computer is all about. So, one generates, as part of the rule base, certain data gathering routines which are run at the start of the session. These routines support certain quantitative rules built into the rule base. Using these routines, the program builds its raw data base. A later part of the activity will crunch the numbers found in the data base and return a truth value to the rule interpreter. There is no need to ask a question the computer can answer for itself.

The Inference Interpreter

What, then, is a knowledge-based expert microcomputer system? It is simply a rule interpreter that has been designed to allow experts to encode their expertise into rule statements - the knowledge base, or rule base. This rule interpreter is capable of reasoning - through the use of the rules, using combinations of mathematical algorithms and a data base, and facts it stores for itself.

What is a "fact" the program stores for itself? Anytime a rule has triggered a question and the answer has been returned, the result is a fact. These facts come from assertions made in the rules, or from calculations performed. Aphids have either been seen, or not been seen today. Either case is a fact. By storing that fact, the program need not ask the question again. It has been said to have "learned" this fact. Had it not stored the newly learned fact, we might observe the expert to have an incredibly short memory. Not too smart. Figure 1-1 illustrates how the knowledge-based expert program links its rules, data base, and facts file to achieve the deductions and conclusions it is chartered to produce.

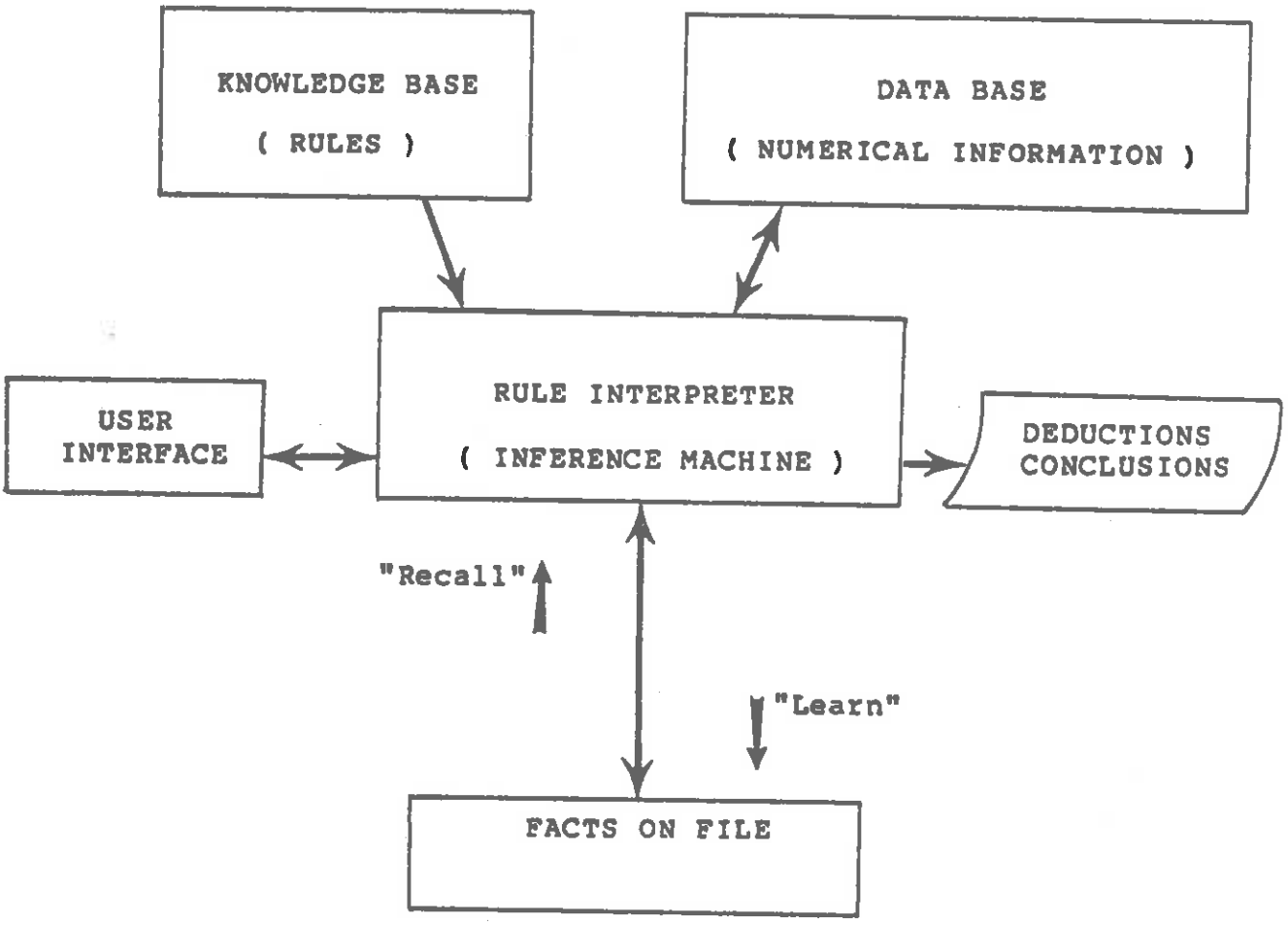


FIGURE 1-1 EXPERT SYSTEM BLOCK DIAGRAM

How does the expert rule interpreter reason? The short of it is that the interpreter plows through the rules testing assertions, checking data, and asking questions until either it knows enough to conclude something, or collapses trying. The long of it is that two methods may be used -separately, or in combination to perform the task:

- Using some data item to trigger a search through the rules - called antecedent reasoning.

- Using an hypothesis - an assumed goal - and trying to find enough facts to form a proof - called consequent reasoning.

If one has a rule base which asserts that:

You have 2 apples	(antecedent 1)
And you have 2 oranges	(antecedent 2)
Proves you have 4 pieces of fruit	(consequent 1)

And one uses a known bit of data - you happen to have 2 apples that you told the system about, the data-driven antecedent reasoning method starts with the data item (2 apples in your possession) and begs to ask if you also happen to have 2 oranges. It's not yet concerned with the consequent it is trying to prove. Figure 1-2 illustrates how antecedent reasoning works.

Assuming an hypothesis "you have 4 pieces of fruit", and without prior knowledge of just what you actually have, consequent reasoning will have the interpreter ask, or otherwise try to prove that, first, you have 2 apples, and last, you have 2 oranges. If it does, it will issue the astounding conclusion that you have 4 pieces of fruit. Figure 1-3 illustrates this consequent reasoning process.

While antecedent reasoning starts with a bit of information and tries to find something to do with it, consequent reasoning starts with the consequent, selected either at random from a list of possible theorems to prove, or by design - as under your instruction (e.g. "TRY consequent 1") or under direction of the rule interpreter.

The Expert System

What does it take to create an expert system? That problem is not unlike the task of creating any high level language.

First, decide what you want the language to be able to do - perform math, test conditions, store information, retrieve information, and conserve with the operator.

Next, decide what you want the vocabulary to look like. This is the process of defining the syntax of the language. In the case of the expert system, it is, more importantly, the process of defining how you wish to

1. Start with fact

Have 2 apples

2. Find a rule to apply

If you have 2 apples
And you have 2 oranges
Then you have 4 pieces of fruit

3. Try the rule

Do you have 2 oranges? Yes or No

Answer = Yes

4. Apply the result

I conclude you have 4 pieces of fruit

FIGURE 1-2 ANTECEDENT REASONING

---

1. Start with hypothesis

You have 4 pieces of fruit

2. Find a rule to try

If you have 2 apples
And you have 2 oranges
Then you have 4 pieces of fruit

3. Try that rule

Do you have 2 apples? Yes or No

Answer = No

4. Apply the result

Cannot prove the hypothesis "you have 4 pieces of fruit"

FIGURE 1-3 CONSEQUENT REASONING

represents knowledge. For some expert systems, representing knowledge is the major design task. As it turns out, there are several ways to represent knowledge; the study of knowledge representation occupies a great deal of the study of artificial intelligence. We will use a syntax which imbeds knowledge in a linked list of IF-THEN constructs for our expert program later.

Finally, the process of creating an expert system includes giving it the capability of learning as it determines the truth value of an assertion. In our expert system here, assertions will either be determined to be true, or false. Either way, once the truth value is determined, the assertion just tested is now a known fact. Our expert system maintains a "known true" file, and a "known false" file.

The expert system we discuss here is a small version of a consequent reasoning system typically used in artificial intelligence work today. Our expert system consists of the knowledge base - the rules - from which conclusions are drawn, a rule interpreter that can perform on both qualitative and quantitative data, and a "facts on file" data base the interpreter builds as it learns.

Our interpreter is also able to offer explanations for its actions. These explanations will serve to teach the user about the subject matter on which the computer is currently acting as expert. Explanations also help us debug our rule base during the codification of knowledge.

CHAPTER 3

A Look at Knowledge Engineering

Expert systems are often used to process knowledge about how humans perform tasks, like diagnosing medical problems, trouble shooting home computers, or identifying minerals for geological exploration. Expert programs could just as well be used as "front-end" processors for configuring large data-base programs for accounting, investment analysis, or, say, animal breeding. No matter what an expert program is used for, its rules form the body of knowledge on which it reasons. These rules also serve to limit the scope of the scene in which the rule interpreter performs the reasoning task.

A global scene, one in which all the knowledge humans possess is codified into rules, is far and away too large a scope for computers as we know them. However, scenes have been created in which global scale activities are limited to specific domains, for example, for testing war strategies. By carefully limiting the scope of activity, it is possible to create a microcomputer-based expert system that can function within the somewhat limited memory space and execution speed of today's home computers. Creating this expert system is called Knowledge Engineering.

The domain we select for our expert program to operate in will determine just how large the scope of knowledge an expert activity can be. To select a domain for our expert, we need to know if the tasks we pose for the expert to consult can be stated as condition-action sequences - IF-THEN statements. If all the knowledge within a domain can be codified as IF-THEN statements, then we limit the scope of our rules to those which satisfy specific requirements we place on the consultant - our expert program. Domains for knowledge representation tend to be classified by three separate characteristics:

- Relative diffusion of knowledge. Medical knowledge tends to be diffuse, with many facts, while physics and chemistry knowledge tends to be concise.

- Inter-dependency of situation-action. An accounting program tends to run with many dependent subprocesses, as does a home computer trouble-shooting program. An expert seismic monitoring program might need only a few dependent subprocesses. If the sensor registers, sound an alarm.

- Separation of knowledge from how it is used. A biological classification program uses knowledge to reason and to gain new knowledge, while an expert consultant program that helps out in the kitchen might only use its knowledge to tell you what to do next.

In developing an expert rule program, it is necessary to specify what knowledge the program will have - its scope - and how that knowledge will

be expressed. The inference machine presented here uses semantic networks comprised of IF-THEN statements as a means of expressing knowledge. For example, if we happen to be privy to the fact that wind blowing from the south, combined with low, dark clouds always precedes rain, we can transform that knowledge into the rule:

```
IF wind is blowing from the south
AND there are low, dark clouds
THEN it will rain
```

Such a rule actually predicts a situation. We might need such a predication to allow us to, by reasoning, offer an expert piece of advice:

```
IF it will rain
THEN close the car windows
```

The advice, "close the car windows" is offered only after our expert program reasons that it will, in fact, rain. Using another example, we ask the computer how fast a stone will be falling at the end of two seconds. We write an expert problem solving program. We know that quantitatively, the stone will gain speed at the rate of 32 feet-per-second for each second it is falling, if we neglect air friction. We know qualitatively, that the stone will fall only if it is unsupported. Our expert ought to know that too. We can write procedures as FORTH functions and allow our expert to use them. This example can be written as follows:

```
: GET/PRINT-DATA
   PAGE CR ." Input time in seconds : "
   QUERY BL WORD NUMBER
   CR CR    ." Speed at end of " DDUP D.
            ." seconds = "    32  1  M*/  D.
            ." feet-per-second. "
   CR CR    1  ;

RULES

IF stone is unsupported
ANDRUN GET/PRINT-DATA
THENHYP stone will fall

DONE
```

ANDRUN is one of a class of operators for the expert interpreter. The operators allow the execution of FORTH functions. These functions may be mathematical computation, user inputs, data-base, etc.

The lone rule listed above is triggered by the hypothesis "stone will fall". The expert must first determine if the stone can fall (because it is unsupported). If the stone is unsupported then the expert performs calculations, to tell us just how fast the stone will travel at the end of the number of seconds we enter.

By using the qualitative reasoning as a front-end to quantitative calculation, we effectively predict what will happen in our stone scene. This qualitative reasoning limits the scope of our expert's reasoning process by determining whether or not to execute certain other rules or procedures - in this case, the equations. If we were to develop an expert program for modeling the performance of, say, a car travelling along the highway, then we might use qualitative reasoning to change the current domain of operation. For example, our computer model might include equations for the performance of the car in scenes which have the car going uphill, downhill, along flat roads, in rain, snow, and so on. Each different scene carries different equations. Each scene is selected by the expert program using its reasoning capabilities derived from the rules we write.

Some Simple Knowledge Engineering Examples

Here are three different "expert" projects we can use for the discussion of our expert system. The first - an animal identification program - is chosen because two different readily available references use this example. These references are listed in Appendix and will be very useful for readers who wish to go beyond the level of this document in developing expert systems. The second example - a tiny stock market assessment tool - takes the ideas we develop and ties them up in an expert program that serves as a useful example of how mathematics, and assertions can be combined to form a powerful consultant. By useful example, it is meant that the expert program developed here serves as an example useful in illustrating how an expert program is put together. By no means is this tiny stock market program represented as useful at making money on Wall Street - or any street, for that matter. Development of money-making expert programs is an exercise left to the reader. The third example diagnosis faults in a digital circuit.

An Animal Expert

Suppose you query a zoo-keeper long enough to find out how he or she identifies some of the animals at that zoo. You are keeping good notes, and, on later reflection, conclude that certain rules can be formed out of what was said. Since the scope of this expert's animal domain is limited to those animals found in a particular zoo, the process of developing expert rules is already simplified; you don't need to test your expert's reasoning talents on animals you have not seen. You plan, however, to use

general animal knowledge you have for yourself to help in the rule design. For example, you note that body hair is an indication that the animal is a mammal. You state that as a rule for you expert system:

IF animal has hair
THEN animal is mammal

You further note another test for a mammal concerns nursing offspring, from which you write another test for a mammal:

IF animal gives milk
THEN animal is mammal

An assertion is made that a mammal is identified if it either "has hair" or "gives milk". Notice that you might have developed a different rule that identifies an animal as mammalian if it "has hair" and "gives milk". That rule might be stated:

IF animal has hair
AND animal gives milk
THEN animal is mammal

But you realize that this test will fail with some hairless mammals -which, by the way, nurse their young. You settle for the first two rules which combine with an "or" implied.

Listing all the animals you learned to identify by the various rules you are developing, you list an albatross, penguin, ostrich, zebra, giraffe, tiger, and cheetah. You note that the albatross, penguin, and ostrich are birds. The zoo-keeper said they were not mammals. So, you develop some rules which support the assertion that the animal is a bird. Birds have feathers, lay eggs, sometimes fly, sometimes even fly well (albatross). You sort these facts into rules which support the bird test.

IF animal has feathers
THEN animal is bird
IF animal flies
AND animal lays eggs
THEN animal is bird

Here you have developed two different rules for defining a bird. You leave them linked together with an implied "or" because the flight characteristic will not support a penguin. Had you included "flies well" in the bird test, you would only allow an albatross to pass the test as a bird. The ostrich would drop out straight away. Now that two tests of "birdness" have been established, you can write rules which support the three different types of birds you have listed. If the animal is a bird which flies well, it is an albatross, and not one of the others. If the animal is a bird, but does not fly at all, it cannot be an albatross, but could be a penguin, or an ostrich.

You note that an ostrich has a long neck, but a penguin does not. A penguin swims, but nothing in your notes supports the idea that an ostrich swims; you choose to ignore that issue because you can probably nail the ostrich on the basis of its long neck. The rules for ostrich are written:

```
IF animal is bird
AND animal does not fly
AND animal has long neck
AND animal is black and white
THEN animal is ostrich
```

We can now see something interesting about the evolution of our rule interpreter. The rules we have developed here are written in a high level language, the syntax for which allows three different operators: IF, AND, and THEN, each followed by a character string that forms the assertion. At this time, it seems likely that the interpreter will spend a lot of its time performing string comparisons - that is, comparing a character string assertion (e.g. "animal flies") to a group of strings it knows to be true or false, or to other groups of strings during the process of finding rules which support whatever it is trying to prove.

Notice that, in our various tests of birdness, we assert that the "animal flies" as one of the tests. If our expert program does not include tests or rules which support "animal flies" (e.g. IF animal flaps wings THEN animal flies), our rule interpreter will likely eventually have to cave in and ask us if the animal flies. If we answer yes, it will know the animal flies. If we provide, in our rule interpreter, the means for it to remember this fact, the newly-learned fact can be recalled later. As it turns out, later happens in our ostrich test, except that we assert there that the "animal does not fly". If our rule interpreter is designed to parse statements, and the parser notes that "does not" removed from the assertion leaves it a negative context of "animal flies", we will not have to put up with the expert asking if the animal does not fly. It will already know it does fly, so why ask again?

If, on the other hand, our rule interpreter is written to perform literal string comparisons only, without benefit of lexical analysis (trying to figure out what was asserted), then the interpreter must eventually ask if the animal does not fly, even if it knows (by way of its facts on file) that the animal flies. So, we introduce another operator: IFNOT, and ANDNOT. These allow us to repeat assertions, but test them for a false condition. Our ostrich test will then look like:

```
IF animal is bird
ANDNOT animal flies
AND animal has long neck
AND animal is black and white
THEN animal is ostrich
```

ANDNOT permits us to work with one assertion about the flying capability of the animal. Literal string comparisons will now work if we now

allow our fact to be remembered in its correct context. That is, it must be remembered as a TRUE fact, or a FALSE fact, depending on what the rule interpreter finds. We have defined the need for two different files of facts. Answer YES to the question "does the animal fly" and that fact is saved in the TRUE file. Answer NO, and that fact is saved in the FALSE file. Either way, the expert "learned" from the question.

A complete listing of the animal rules is provided in Appendix 2. This set of rules can be used later to test our rule interpreter.

We have just developed a simple syntax for our high level language expert system. Our next example problem, the stock market analysis "expert" will add to this syntax to yield the final requirements for our rule interpreter.

A Stock Market Expert

Suppose you need an expert to tell you whether the stock market is in a bull, or a bear market. You decide to define a bull market as one in which you ought to consider buying, and a bear market as one in which you ought to consider selling. Actually, you want to sell out at the peak -the end - of a bull market, just before the bear market starts, and you want to buy at the bottom of the bear market, just before the bull starts another run. You're not alone in your desire.

It would be an incredibly large task to develop an expert program that combines all expertise available for successful investing - even when the field is limited to a stock market. It is reasonable, however, to create expert programs which offer expertise in domains which are carefully limited and selected to provide maximized return on the time invested in creating the program. Limiting the scope of an expert program is one of the maxims of knowledge engineering; don't try to create omni-knowledgeable expert programs.

So, you decide to assign the task of spotting the ends of bull and bear markets to your trusty home computer. You want to give it an expert program which will use, as its data base, information you get from your broker at the end of each trading data. It will also use a rule base created by you, based on your experience, research, and prying from successful friends. Here's a sample of how that might go, limited, here, to observing the New York Stock Exchange.

In your research, you discover what you believe to be a nifty way to track the "power" in the day's trades. You want to be able to determine, as one of your expert tests, whether today's trading on the NYSE is stronger (more bullish) than the previous day's trading. You might settle for simply comparing trading volume, and closing Dow Jones Industries level, but you think this algorithm that you discovered in a book might be better. You decide to include it in your expert. It's a "climax breadth" test that seemed to emerge during the 1920's. You ignore the great bear

that also emerged back then.

A cumulative scaled climax breadth index is maintained. Each day the advances less the declines divided by the total traded and scaled by a factor of 100 is added to the previous value.

In order to make this calculation, you need to enter the three parameters. You also need a new class of operators for your expert rule interpreter. Let's define the new operators: IFRUN, ANDRUN, and THENRUN. This class of operators lets us run any FORTH function which has been defined. Such functions might include the operator inputs, mathematics, and data base requirements you define in MVP-FORTH.

In this application, we need to update the climax breadth index if we have the data. The function of START, provides the necessary prompts for the required information and performs the actual operation. When executing DIAGNOSE with this rule base, the user is first asked if there is new data. A true flag is left whether there is or not and the analysis proceeds. The other functions are simply the tests which are used in the rule base.

```
: START
   PAGE  CR  ." New data ?  <Y/N> "  KEY
   CR  CR  89 ( Y ) =
   IF CR ." Input stock advances: " INPUT#
      CR ." Input stock declines: " INPUT#    D-
      CR ." Input stocks traded:  " INPUT#
      DROP  100 SWAP M*/ DROP  CB  +!
      CR  CR  ." The new market strength = "  CB  @  .
      CR  CR
   THEN    1  ;

: CB?
   CB  @  Ø >  ;

: BULL?
   CB  @  1ØØ >  ;

: BEAR?
   CB  @  -1ØØ <  ;
```

It will be necessary to define the variable CB and the function INPUT#. The full program is included in the Appendix. By accumulating the new values on the stack and making the calculations as the data is input, a minimum of additional variables is necessary. Each of these FORTH functions leaves a truth flag on the parameter stack for the rule interpreter.

Now, you examine another area of market activity - trying to guess what the insiders are doing. If they are fixing to sell, you want to sell, too. If they are buying - accumulating stocks at wholesale prices with

which to make a killing by selling at retail, you want to buy, too. You observe that a bull market run-up of stock prices seems to end with a large increase in trading volume, but little or no increase in price. Occasionally, market strength, as described by climax breadth, weakens, or goes negative. You further observe that a bear market or "correction" seems to end with a noticeable jump in volume without a great change in DJI prices, and not much change in market strength. You decide, at your Saturday investment club meeting, that insiders stop a bull market with massive short-selling (selling stocks they sometimes don't have) which means they have dumped their stock at retail, and are ready to start accumulating again, after the stocks drop again for a while. You speculate the insiders make money either way, because, by short-selling at the top, they replace what they sold later, at lower prices, netting them another killing. In fact, you further conclude that it is this buying at the bottom to replace stocks sold short that signals the end of a bear market. So you've invented a set of rules you want to use with your expert computer.

Based on this analysis and the functions listed above, we can develop the rule base necessary. As with any EXPERT SYSTEM rule base, the results will be no better than the analysis upon which the rules are based. In the sixty some years since this analysis was first published, no one has successfully exploited it. Be careful that you only consider this as an example of the development of a rule base.

The conclusions are included in the rules which end with THENHYP. They are:

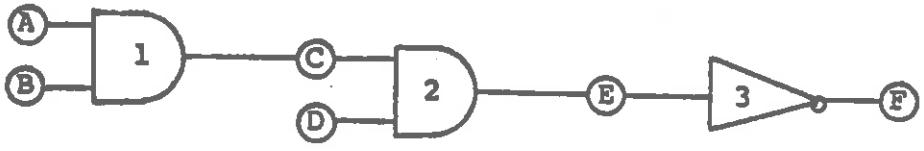
- o indications are to buy
- o indications are to sell
- o do nothing -- wait

This new class of operators simply executes FORTH functions. These amount to a group of procedures as they might be called in other languages. The ---RUN is a carry over from languages which run procedures. Each of these FORTH functions may be executed. They must, however, leave a truth flag on the top of the stack after all other activities are completed.

It is unlikely that any money can be made using those rules. They do, however, illustrate a rule generation process, and a separation between rules which govern decisions, and rules which guide analysis of data.

A Digital Circuit Analyzer

You are confronted with the need to develop an expert system to analyze faults which might occur in the digital circuit shown below. Using the expert system tools we have discussed already, it is possible to design a short program that will isolate which of the three circuit elements has failed. Variations, and improvements on this simple program are, of course, possible.



Simple Digital Circuit

Chip 3 is an inverter which has two possible modes of failure: its output (F) will be struck true, or it will be stuck false. Diagnosis involves measuring the input state with a voltmeter, scope, or logic probe, and measuring the corresponding output state. An an inverter, its output must be the opposite of its input. Two rules can be written to detect a faulty inverter:

```
IFNOT E is true
ANDNOT F is true
THENHYP Chip 3 is bad (input and output false)

IF E is true
AND F is true
THENHYP Chip 3 is bad (input and output true)
```

Chips 1 and 2 are AND gates, with a greater number of failure modes. More rules will be needed. Using Chip 2 as an example:

```
IFNOT C is true
AND E is true
THENHYP Chip 2 is bad (output true, 1 input false)

IFNOT D is true
AND E is true
THENHYP Chip 2 is bad (output true, 1 input false)

IFNOT E is true
AND C is true
AND D is true
THENHYP Chip 2 is bad (output false, both inputs true)
```

A similar rule can be written for Chip 1. When a session at the computer runs this diagnostic program, the inference machine will select one of three possible hypotheses:

```
Chip 3 is bad
Chip 2 is bad
Chip 1 is bad
```

The selected hypothesis will then face a proof test. Along the way, the inference machine will learn various facts. For example, the system might choose to diagnose Chip 3 first. It will ask:

```
Is this true?
     E is true
```

If the answer is YES, the system must now see if F is also true. An an inverter, F must be made false. The computer asks:

```
Is this true?
     F is true
```

If the answer is NO, Chip 3 cannot be proven bad. The expert system now knows, however, that E is true and F is false, for whatever those facts are worth to subsequent tests.

The system now chooses to prove Chip 2 is bad. Remember it already knows that E is true. With regard to Chip 2, there is one rule which asks IFNOT E is true. This rule automatically fails, leaving two possible tests:

```
Is this true?
     C is true
```

If the answer is YES, the system must now opt for the last rule:

```
Is this true?
     D is true
```

If the answer is NO, the system will conclude "Chip 2 is bad".

With care and experience, it is possible to generate a set of generic rules describing inverters, AND gates, OR gates, and other chips. With these generic rules, one can string together diagnostic programs for just about any circuit.

This expert program can also be written to use analytical subroutines to control automatic testing of the circuit. This would eliminate the need to ask a technician all the necessary questions. Such a program, after checking readings, would deduce a failed chip and ask the technician to confirm the deduction.

Functional source code for these example expert programs is included in Appendix B and on the distribution screens diskette.

With these example expert programs in mind, we now look at the syntax and internal design of EXPERT-2.

Expert Systems and the Weather

by Jack Park

Somebody famous once said: "This is gonna get me in trouble." Since I am about to discuss a program that is capable of predicting the weather (sometimes), I figure I'll get in trouble, but here goes.

I chose predicting the weather as the subject of an early effort at knowledge engineering using EXPERT-2, a tiny fifth-generation type language. My friends in the business of weather prediction think this is somewhat amusing, but the results are quite interesting and possibly useful. I deliberately kept the knowledge engineering exercise simple since I intended to run this program in a 48K Apple II, using a Forth system with EXPERT-2 on top and the application program, the weather predictor.

Expert systems are computer programs that are specially designed to perform inferences: to prove things. EXPERT-2 is a program that allows individuals to write task programs much like BASIC lets users write programs. It is a high-level language written for performing logic inferences using rules written into the user's program. These rules may be used to solve important problems, like predicting the stock market . . . or the weather.

EXPERT-2 allows two methods for encoding knowledge and information into the task program — in this case, the weather predictor:

IF-THEN statements

Analytic subroutines

IF-THEN statements, or production rules, allow the user to direct the inference process by entering into the computer specific statements that say, "if something is true, then something else is true." The EXPERT-2 syntax is an extension of the IF-THEN syntax used to develop an animals game.[2,3] The extensions include allowing statements to be entered in a negative context (IFNOT, ANDNOT) and calling analytic subroutines.

Analytic subroutines were added to give the programmer access to the full power of the underlying Forth system. If EXPERT-2 had been written in, say, Pascal, then the analytic subroutine calls (IFRUN, IFNOTRUN, ANDRUN, ANDNOTRUN) would permit access to that environment. As it is, EXPERT-2 was written in Forth, primarily because that's the program environment in which I work best. Users of EXPERT-2, however, need not know or understand Forth; if analytic subroutines are not needed, no Forth coding is required.

The weather predictor presented here does use analytic subroutines. These routines are used in two ways:

To prompt the user to input data

To process data and return truth values to the rules

The IF-THEN rules call the appropriate analytic subroutines when answers about weather data are needed.

The IF-THEN rules contain knowledge about the weather. This knowledge is encoded as antecedents (e.g., IF barometric pressure is falling rapidly) and consequents (e.g., THEN the weather is turning bad). Consequents follow appropriate antecedents. Designing rules to correctly encode knowledge is called knowledge engineering.

Knowledge Engineering

Someone once asked me if EXPERT-2 ever adds to its own knowledge base or makes original statements. As it turns out, it often makes original statements – especially during system debugging. Early versions of EXPERT-2, in fact, were self-modifying (not by design!). Test runs were incredibly mystifying. Knowledge engineering, as I use the term here, assumes one has available a working inference machine: a program that permits inferences to be performed on a set of rules. EXPERT-2 is one of the early microcomputer-based inference machines available; I expect there eventually will be a slew of them.

Knowledge engineering is the process of defining a problem to solve (something for the expert to do, especially something useful), encoding into a program the expertise required to solve the problem, testing, and finally validating the expert program.

To encode knowledge, one needs an expert whose brains (expertise) are accessible for encoding. As it turns out, that's not a generally available commodity; experts may be around, but it takes charm, wit, and experience to get at their brains. Trying to determine what tools an expert brings to bear on a problem often gets you a textbook set of answers. Textbook answers are not necessarily what you need to solve a problem. Factors including years of experience, intuition, and a host of other issues must be explored before a knowledge engineer fully exhausts all the elements needed to solve a particular problem.

For the weather predictor, I chose to explore only books and magazines, partly because I didn't want to call in any chits I may have with the weather service and partly because I discovered some interesting prospects in the literature for homebrew weather prediction that could be translated to the EXPERT-2 system. A set of programs have been presented for computers that run BASIC.[4] These programs ask certain questions and issue weather predictions based on the answers. Because the algorithm used in the programs relates well to the general literature on weather prediction, I selected it as a representative "expert" set of rules.

For additions and changes to the rules derived from the magazine article I drew on the Heathkit Weather Training Manual, which has a section on weather forecasting.[5] Selected rules on cloud motion, for example, were used to enhance the rule base.

Analytic subroutines are then used to acquire weather data from the user and to process that data as the inference machine needs answers.

Weather patterns are quite sensitive to local terrain influences. Therefore, the responses the weather predictor offers may not be appropriate to, say, a canyon region where you live. It seems to work well where I live. I leave it as an exercise for the reader to validate the weather predictor program in his or her own area.

System Operation

Rules are typed into a disk file using the editor available with the underlying Forth system. EXPERT-2 is then loaded and compiled on top of Forth as a task. Finally, the user task (in this case, the weather predictor) is loaded on top of EXPERT-2. First Forth compiles the analytic subroutines and any variables or constants. EXPERT-2 then compiles the IF-THEN rules once the word RULES is encountered. Rule compiling ends with the word DONE.

Forth, in its usual fashion, tells you that compiling has ended by printing OK on the screen. At this time it should be

safe to type RUN and start the forecasting program. RUN initializes the data base by asking appropriate questions then calls EXPERT-2 to solve the problems. The problem is to prove one of four possible hypotheses, identified in the rule base by THENHYP. The four hypotheses are:

Weather is OK

Weather is improving

Weather is turning bad

Insufficient data to perform a forecast

EXPERT-2 takes the first hypothesis and begins the task of proving it. The routine that performs this task is DIAGNOSE, which is called by RUN. DIAGNOSE selects the hypothesis and passes it deeper into the EXPERT-2 program. EXPERT-2 collects all the rules that might support a proof.

If the hypothesis is "weather is OK," then all rules that have as one of their consequent fields "weather is OK" are tagged and drawn into the inference process. One such rule is:

```
IFRUN BP>30.2
ANDRUN BP-SLOW-FALL
ANDRUN WDIR
BECAUSE WINDS FROM SW, W,
    OR NW
THENHYP WEATHER OK
ANDTHEN FAIR AND WARM
    NEXT 48 HRS
```

One of the consequent fields (in this case, the first consequent) is the same as our hypothesis; in fact, this rule happens to be the one selected to define an hypothesis with THENHYP. EXPERT-2 finds this rule and tries to prove it.

The inference machine takes each of the antecedents, one at a time, and tests them. If all pass with a truth value of TRUE, the consequents are taken as true, and the proof is complete. If any antecedent fails with a truth value of FALSE, the rest of that rule is not tested. Anything that is "learned" along the way, however, is saved as a fact. If the system asks you which way the clouds are moving, for example, your answer becomes a fact so that EXPERT-2 does not have to ask you that question again.

If EXPERT-2 proves a rule, DIAGNOSE tells you about it, and the program ends with I CONCLUDE . . . . If, on the other hand, a given hypothesis cannot be proven, then the next available hypothesis is selected, and a new proof is started. The new proof has available to it everything that the system learned during all previous proof attempts (all attempts, that is, since RUN was typed). If no proof is available on any hypothesis, EXPERT-2 typically states, CANNOT PROVE ANYTHING.

The Weather Predictor

You may wonder how such a simple-looking program can do any useful weather prediction; most people ask about this. As it turns out, the program concerns itself only with large-scale frontal movements. These fronts are best visualized (but not correctly described) as large bubbles of air hundreds of miles across, sliding around over the ground, bumping into each other, and generally moving from the west toward the east. Some of these bubbles are huge low-pressure masses of air, and some are large high-pressure masses of air.

When a low-pressure mass of air approaches your area, the barometer will spot the frontal motion and indicate that the barometric pressure is falling. As that air mass leaves your area, the barometer will begin to rise. Thus, by using readings of barometric pressure and asking questions about which way the winds are blowing, the weather predictor attempts to determine which of its "known" frontal-air-mass patterns best describes the conditions present in the atmosphere. If it finds a pattern that fits, it issues a prediction. The patterns it looks for, however, must be calibrated for your area.

As a simple exercise in knowledge engineering, the weather predictor demonstrates nearly the full capabilities of EXPERT-2. The program uses information on barometric pressure, surface wind direction, and upper air wind direction (as evidenced in upper cloud motion).

The rules begin with WALL, a word that lets you clear this rule base from memory if you want to load another program (by typing FORGET WALL). Following WALL, all analytic subroutines are loaded, written in Forth. This version (Listing One, page 27) is written in an extended Forth-79 dialect and based on the 40-character line width of a standard Apple II. Readers who type this in on other systems are encouraged to explore the enormous advantages of larger line widths; some awkward statements often are created when one is constrained to a narrow screen.

Following the analytic subroutines, RULES starts EXPERT-2's rule compiler. This invokes the IF-THEN syntax that the listing illustrates.

This weather predictor tests for 19 different possible combinations of weather conditions, using 19 basic weather prediction rules. If the requested weather information you type in fits within one of the 19 possible data combinations, a prediction will be issued. If the data falls outside all possible tests, the system will issue the statement INSUFFICIENT DATA FOR A FORECAST.

Since EXPERT-2 would rather say CANNOT PROVE ANYTHING, the last rule in the set stops that statement and issues a custom proof-collapse statement. That way, if EXPERT-2 is unable to prove any of the three main hypotheses, it is offered the last hypothesis as a consolation prize. This allows you to end an unsuccessful session with any statement you wish.

Three sets of rules are available for checking on upper level clouds. These rules, which support some of the 19 main rules by asking special questions about such clouds, underscore one of the weaknesses of EXPERT-2: its inability to handle an "I don't know" answer. That answer, in fact, does not exist in the syntax. An astute knowledge engineer, however, is still burdened with the likely event that this particular answer will be needed. Weather predictor handles the "don't know" case by allowing you to answer N for no if either you cannot see the upper clouds (there may not be any clouds to see) or you haven't gone outside to look as instructed. If you answer no, the system assumes the cloud motion supports whatever rule it happens to be trying to prove at the time.

This derives from the second rule in each of the three rule groups. The second rule, in effect, says: After dealing with the instructions (MESSAGE1), IFNOT you can see the clouds THEN clouds indicate . . . . One of these exists for each of the three possible conditions. A true condition is forced if you answer no. I am quite sure I'm gonna get in trouble for this.

A typical terminal session with this weather predictor follows:

1. System starts by asking barometric pressure; you enter 30.2.
2. System next asks what the barometric pressure is doing; you enter the code for falling slowly.
3. System asks for the direction from which the wind blows; you enter the code for south.
4. The terminal response is:

```
NOW, WE CAN PREDICT THE
WEATHER.
I MIGHT BE ASKING A FEW MORE
QUESTIONS.
I DEDUCE
    RAIN WITHIN 24 HRS
I DEDUCE
    WEATHER TURNING BAD
I CONCLUDE
    WEATHER TURNING BAD
OK (Forth's usual prompt)
```

No further questions were asked in that session. This particular session included data that triggered (successfully) the following rule:

```
IFRUN BP>30.1
   (30.2)
ANDRUN BP-SLOW-FALL
   (pressure falling slowly)
ANDRUN SDIR
   (southerly direction winds)
```

```
BECAUSE WINDS FROM SE OR S
THEN RAIN WITHIN 24 HRS
ANDTHEN WEATHER TURNING
  BAD
```

This rule illustrates a number of the design methods that were used in this knowledge engineering exercise. First, the final consequent, "weather turning bad," helps direct the proof search. If you read the rules, especially the consequent fields, you will see that a search could be conducted on dozens of different possible consequents: "rain within 24 hrs," for example. If I had set as many hypotheses as there are different consequents in this program, the hypothesis field would have been enormous. Instead I chose three primary consequents as "search pruners" to limit the search requirement. All possible rules that have a particular hypothesis are collected at the same time for proof. The first rule to pass the test terminates all further tests. Other than that, "weather turning bad" is a pretty useless consequent.

A BECAUSE field is included in this rule to allow the programmer to tack explanations in where the code might otherwise be incomprehensible. For example, if the name chosen for an analytic subroutine that is called to check wind direction is not particularly descriptive, a BECAUSE statement is added. If the system asks a question in the form IS THIS TRUE?, you can answer Y for yes, N for no, or W for "Why did you ask me that question?" It allows you to get an explanation, which is where the BECAUSE clause is handy.

The BECAUSE clause is also handy when passing around source code for programs. One of the advantages of the EXPERT-2 style program is that it is largely self-commenting. Where it is not self-commenting, you can add BECAUSE clauses.

This EXPERT-2 program's heavy use of analytic subroutines reduces the number of questions the program might be inclined to ask at the terminal. By contrast, the animals game mentioned earlier uses absolutely no analytic subroutines: whatever it learns in the proof process is gained from the terminal. Expert programs are much more interesting if they ask something at the terminal.

Going Beyond

EXPERT-2 is a high-level language for experimenters in early fifth-generation machines. It is made available in source code form, loadable on a variety of Forth systems. Using the source, this weather predictor program, and a lot of sweat equity, I expect some pretty strong weather predictor programs will emerge.

Weather prediction is largely a data processing and pattern recognition effort, one well suited to an expert system. By adding some simplified models related to atmospheric physics and coupling the program (through analytic subroutine functions) to an automatic weather station, you could perhaps create a new program for predicting weather hazards for farmers or sporting events.

Weather forecasters rate each other by a value they call "skill." Flipping a coin usually gets a skill of 0.5 or 50% right. It will be interesting to see what skill this program and its derivatives achieve. Since local weather prediction is sensitive to local geographic terrain effects, this program will need some fiddling to raise its skill, perhaps even to the 50% level.

Some thought might go into letting the program run on a pseudo-real-time basis, keeping track of its own skill. It also might be written to fiddle with its own inferences in an attempt to raise its skill: a self-learning system. You will have to make sure, however, that it does not learn how to fiddle with its skill algorithm.

[2] Richard Duda and John Gaschnig. "Knowledge-Based Expert Systems Come of Age." *Byte*. September 1981.

[3] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley.

[4] George W. Miller. "Weather Forecaster." *COMPUTE!* August 1983. See also Tom Fox, *Unique Electronic Weather Projects*. Howard Sams & Co., 1978.

[5] *Weather Training Manual*. Benton Harbor, MI: Heath Company.

CHAPTER 4

EXPERT-2

Any expert system has three components:

- o a knowledge base - the expert program
- o an inference machine to run the expert program
- o computer hardware

Each of these components is discussed here. We start with the computer hardware environment in which EXPERT-2 runs.

Computer Hardware

Just a slight amount of literary license is needed to state here that the computer hardware environment in which EXPERT-2 runs includes both the computer system itself, and the MVP-FORTH language compiler/interpreter. By writing EXPERT-2 in MVP-FORTH, the resulting inference machine is transportable to a large variety of desk-top computers.

The computer system is required to have at least one disk drive, large enough memory for the MVP-FORTH language, EXPERT-2, and the expert program to be run by EXPERT-2, and a terminal for communication with the user.

The Expert Program

There are two program components available to you with EXPERT-2:

- o IF-THEN rules
- o analytical subroutines

IF-THEN rules are created using a special syntax which is fully described below. These rules can use word statements, or can call analytical subroutines. In either case, truth values are manipulated. Truth values are binary in EXPERT-2; a statement or result of analytical calculation can either result in a TRUE or a FALSE condition.

Analytical subroutines are written in MVP-FORTH. By using MVP-FORTH as the language for all analytical subroutines, the MVP-FORTH compiler/interpreter underlying EXPERT-2 is available for subroutine work.

A specific format is required in planning an expert program for execution under EXPERT-2. This format is illustrated in Figure 4-1, and discussed here.

Any expert program should begin with a MVP-FORTH word that has little meaning to the program itself. This word is compiled to serve as a fence,

to provide you with a word to "FORGET" when it is time to load a different expert program. The MVP-FORTH word FORGET clears the program memory space up to, and including the word you choose to FORGET. The word "WALL" is suggested as a standard for EXPERT-2 programs so that all users simply need to FORGET WALL to clear any given expert program from memory.

EXPERT PROGRAM FORMAT

```
: WALL ;     ( A compiled word to FORGET when done )
Compile any analytical subroutines here - between WALL and RULES.
The following analytical word is listed for illustration.

: DEMOWORD CR
  ." ENTER A NUMBER ( 1 TO 10 ) " CR (ASK FOR NUMBER ENTRY)
  KEY  53        ( GET THE NUMBER AND TEST FOR GREATER THAN 5 )
  IF TRUE      ( NUMBER IS GREATER THAN 5 - RETURN TRUE FLAG )
  ELSE FALSE     ( NUMBER IS LESS THAN 5 - RETURN FALSE FLAG )
  THEN ;

Note: as many analytical words as required may be compiled here.

RULES ( this begins the IF-THEN rules )
( RULE 0 )
     IF we are ready to begin
     ANDIFRUN DEMOWORD
     BECAUSE a number > 5 is needed
     THEN a number > 5 is available

( RULE 1 )
     IF a number > 5 is available
     THENHYP all conditions are satisfied

DONE ( end of IF-THEN rules, end of expert program )
```

FIGURE 4-1 EXPERT PROGRAM FORMAT

Analytical Subroutines

Just after the WALL, space is available for analytical subroutines. This is an optional space; if not analytical subroutines are required - as in, for example, the animals game - no subroutines need be listed. As many subroutines as are required to perform the expert task may be listed, with only one limitation: computer memory size. You must pay attention to available space in your computer system; it is quite easy to get carried away with analytical subroutines and IF-THEN rules and exceed the memory capacity of one's computer.

Development of analytical subroutines only requires that you pay attention to the MVP-FORTH maxim that forward references are not easily performed. That means that subroutines must be compiled before they are called. Subroutines may call other subroutines in the usual MVP-FORTH fashion of using a subroutine's name.

IF-THEN Rules

After all subroutines have been listed, the rule base may begin. In EXPERT-2, the rule base begins with the word RULES. EXPERT-2 recognizes RULES as an executable word in the EXPERT-2 program. RULES sets-up the MVP-FORTH system to read and compile the following rules using the RULE vocabulary. By changing to the RULE vocabulary, it is possible to redefine the traditional MVP-FORTH words IF, AND, and THEN to have an expert meaning, instead of the usual MVP-FORTH meaning.

Rules consist of an operator, such as IF, or THEN, followed by either a character string, or the name of an analytical subroutine.

A physical size restriction is placed on any character string used in EXPERT-2. The lone limitation is length of the string. This limitation is that the number of characters, including spaces and the operator (IF, THEN, etc) not exceed 64 characters. That is the line-width for MVP-FORTH editors. This limitation applies to all rule operators, including IF, ANDIF, THEN, ANDTHEN, and BECAUSE.

EXPERT-2 Syntax

IF <antecedent character string>
    example: IF animal has feathers

ANDIF <antecedent character string>
    example: ANDIF animal flies

THEN <consequent character string>
    example: THEN animal is bird

ANDTHEN <consequent character string>
    example: ANDTHEN animal has wings

These four operators comprise the primary, positive context character string rules. Negative context character string rules for antecedent testing are also permitted.

IFNOT <antecedent character string>
example: IFNOT animal is bird

ANDNOT <antecedent character string>
example: ANDNOT animal swims

IFNOT and ANDNOT are created to permit the same character strings to be used in the positive and negative context. This permits memory efficiency in the compiler, since any given character string is compiled only once, no matter how many times it is used in the rule base, and no matter in which context it is used. Such memory efficiency is required to fit useful expert programs into tiny desk-top computers. Note, however, that a typographical error in any multiple use of a given character string will result in a rule-base error since the string compiler routine will not recognize the erroneous string as one already present in memory. The result of such an error is usually an unproveable rule. EXPERT-2 demands that multiple uses of the same character string be typographically checked by the knowledge engineer.

A contraction of one of the above-listed operators is valid. This contraction makes the operators shorter, providing more room for character strings.

ANDIF <antecedent string> can be **AND** <antecedent string>

Analytical subroutines may be called from the rule base with the following operators:

IFRUN <subroutine name>
example: IFRUN demoword

IFNOTRUN <subroutine name>
example: IFNOTRUN demoword

ANDIFRUN <subroutine name>
example: ANDIFRUN otherword

ANDNOTRUN <subroutine name>
example: ANDNOTRUN otherword

THENRUN <subroutine name>
example: THENRUN consequentword

ANDTHENRUN <subroutine name>
example: ANDTHENRUN anotherconsequentword

ANDIFRUN <subroutine name> may be contracted to **ANDRUN** <sub name>

EXPERT-2 places a rule design requirement on the knowledge engineer related to the use of THENRUN. EXPERT-2 is a backward chaining (consequent reasoning) inference machine. As such, it requires a character string in the consequent field when collecting rules to prove a hypothesis. Thus, the requirement that at least one of the consequents in any given rule must be a character string consequent, using THEN, or ANDTHEN. For purposes of program design, the option exists to simply identify any given analytical rule with a dummy consequent, as, for example THEN XX. Somewhere else in the program, an antecedent, such as IF XX, will trigger this particular analytical rule. It is suggested, as a matter of standard, to refrain from using dummy statements. This is a fine opportunity to use the string as an explanatory statement - what the rule is for, what is to be solved, or the like. EXPERT-2 provides the user the ability to ask why a given question is asked. This provision works by printing the specific rule being tested. Thus, using a dummy consequent string, for purpose of permitting EXPERT-2 to trigger an analytical rule, is also an opportunity to converse with the user.

EXPERT-2 selects an hyposthesis to prove from a stack of hypotheses created during compilation of the rules. In order to reduce the number of times one must type a given character string, the following operator has been created to permit the knowledge engineer to identify any given consequent character string as an hypothesis to prove later.

THENHYP <consequent character string>
example: THENHYP animal is penguin

Using THENHYP, the rule base sees "animal is penguin" as a consequent to be proven, and, also, the inference machine sees the same character string as one of the possible goals it must reach in the development of a proof. Any consequent which is planned to also be a final possible conclusion at the end of a session with EXPERT-2 is identified in its rule by THENHYP instead of THEN.

EXPERT-2 makes provisions for brief explanations. Two operators are available for use in the explanation phase. These operators are not tested or otherwise executed by the inference machine; instead, they become available during the user session when the user asks why a given question was triggered.

BECAUSE <explanatory character string>
example: BECAUSE all birds have feathers

BECAUSERUN <analytical subroutine name>
example: BECAUSERUN DISPLAYDATA

BECAUSERUN is a special operator which permits use of analytical subroutines for the purpose of displaying information in the data base, as

for example, the result of a calculation that had been previously performed. An analytical subroutine is not strictly limited to mathematical operations; any function permitted by the underlying MVP-FORTH system may be used in an analytical subroutine. Thus, for example, BECAUSERUN might be used to call a word which simply prints a larger explanation than is permitted in the normal character string line.

Rules to be compiled after RULES in an expert program need not be listed in any particular order to satisfy the underlying MVP-FORTH compiler. This is because rules are not, in themselves, subroutines which can call other rules. Rules can only call analytical subroutines which have already been compiled. No forward references are created by multiple uses of any given character string, and the rules may be listed in any convenient order.

The order in which rules are listed can affect the speed of execution of a user expert program. Execution speed can be enhanced by careful grouping of rules around the search order they present. If, for example, the most important rules in a given computer fault diagnostic expert program deal with, say, the output of an inverter circuit, one might consider placing those rules first in the rule list. EXPERT-2 starts at the beginning of the rule list when searching for rules to test in support of a given hypothesis. Placing the most important rules first reduces the time required to find them.

The rule-based expert program is concluded with the word DONE. This word ends the rule compiler activity, and tidies-up the rule base.

By way of review, an expert program written to be run under EXPERT-2 starts with WALL, followed by the analytical subroutines written in MVP-FORTH, and finally followed by the rules. The rules are listed following the word RULES, and ended with DONE.

An expert program is written using the editor facility provided by the underlying MVP-FORTH system. Screens of rules are written and saved to disk.

Rules are loaded into the computer the same way any program is loaded. The assumption is that, first, MVP-FORTH has been loaded, and then EXPERT-2 has been loaded. Now, the rule-based expert program may be loaded. If, for example, the expert program starts on screen 100 then one types 100 LOAD. This starts the entire compiling process. The process ends with the usual MVP-FORTH prompt "OK". Execution of the expert program begins by calling the inference machine. In EXPERT-2, type DIAGNOSE to start the expert program.

In the next section, we discuss the internal operation of EXPERT-2. It will be helpful to refer to Appendix 2, the source listing of EXPERT-2.

The Inference Machine - EXPERT-2

EXPERT-2 is the second version of a group of inference machines. This version was selected for release because of its inherent simplicity, and reasonable power. It is released as a tool-kit for individuals who wish to create expert programs, and as an educational tool for individuals who wish to explore inference machines. EXPERT-2 can be simplified, or it can be expanded or otherwise modified to suit individual needs.

****

Note: EXPERT-2 is not released as a public-domain program for wholesale duplication and distribution. It may be modified and used as a basis for the work of others, and is intended to be so used. However, trivial changes, and republication of EXPERT-2 will constitute an infringement of the originator's copyrights.

****

EXPERT-2 is written in MVP-FORTH. In order to fully understand the internal processes of EXPERT-2, the reader must be familiar with the MVP-FORTH language. Starting FORTH, a good MVP-FORTH language tutorial text is listed in the Appendix.

The EXPERT-2 source code is loosely organized from lowest level inference machine support words, to the inference machine itself, followed by the rule compiler.

The rule compiler is responsible for reading an expert program source file from disk, and compiling the rules in memory.

MVP-FORTH/EXPERT-2 MEMORY MAP

ØØØØ Beginning of memory

This area contains various pointers, stacks, and other working space required by the host computer, and the MVP-FORTH system.

vvvv Beginning of MVP-FORTH system

wwww End of basic MVP-FORTH system
Beginning of EXPERT-2

xxxx End of EXPERT-2
Beginning of user's expert program

The first word typically compiled here (at xxxx) is WALL. WALL is followed by any analytical subroutines listed in the source

Analytical subroutines are followed by the rule base.

yyyy End of the rule base
Potential beginning of character strings

All character strings are compiled somewhere in memory. The specific starting location is determined either by (yyyy) the end of the rule base -which can be calculated ahead of time - or by specific allocation.

zzzz End of character strings

FFFF End of typical 64K memory

FIGURE 4-2 MVP-FORTH/EXPERT-2 MEMORY MAP

Analytical subroutines are compiled directly by the underlying MVP-FORTH system. They are compiled into memory just after the EXPERT-2 system. Figure 4-2 is a memory map of a typical EXPERT-2 system.

The word RULES starts compilation of the rule base. Rules are compiled into memory just beyond the analytical subroutines, while the strings used in the rules are compiled elsewhere.

Rules are compiled into cells, one cell per rule. A rule cell consists of an antecedent field, a consequent field, and field markers. Field markers are three bytes of zero (ØØØ). Field markers follow the antecedent field, and the consequent field. Thus, the rule:

```
IF animal has feathers
ANDIF animal flies
THEN animal is a bird
```

compiles into the MVP-FORTH dictionary to look like:

```
Ø ADDR1 Ø ADDR2 ( antecedent field )
Ø Ø Ø           ( end flag )
Ø ADDR3         ( consequent field )
Ø Ø Ø           ( end flag )
```

Notice that each address (ADDR1, ADDR2, and ADDR3) is preceded by a byte-length integer. This integer number is a flag representing the type, and context of the following rule element. A Ø flag is reserved for positive context strings, while a 1 is reserved for negative context strings. Other integers are reserved for analytical equations, and BECAUSE clauses. Notice that expansion of EXPERT-2's capabilities can be facilitated by using other integers to represent expanded rule representations.

The address compiled in a rule element is found by the word GETCLAUSE, if the rule happens to be one of the character string operators. If the particular string has already been compiled, it will be found by FINDCLAUSE, and the address where it is found is passed back to the rule compiler. If the particular string has not already been compiled, it is then compiled into the next available memory space, and the beginning address is returned to the rule compiler, which inserts that address into the rule base it is building.

The inference machine is the word DIAGNOSE. This word simply pops a pointer to an hypothesis character string off the hypothesis stack (called HYPSTACK) and proceeds to attempt to prove that hypothesis to be true. If the hypothesis is found to be true, DIAGNOSE concludes that string to be true - and tells the operator of this conclusion. If the hypothesis is found false, DIAGNOSE pops another hypothesis off the stack and tries

again. In the event all hypotheses on the stack fail, DIAGNOSE tells the user it cannot prove anything, and exits back to MVP-FORTH with the usual "OK".

This is an incredibly simple inference machine, but one which allows some very powerful expert programs to run. DIAGNOSE does not require any more than one hypothesis. Several hypotheses may be designed into the expert program, but the specific number of hypotheses allowed is limited only by the size of HYPSTACK. HYPSTACK, as supplied, allots 256 bytes, resulting in room for 128 pointers. 128 pointers imply at least 128 different consequent strings, each of which implies one or more antecedents. 128 hypotheses, in short, implies, most likely, an expert program much larger than memory space will allow.

DIAGNOSE hides the hypothesis pointer in a variable CURHYP for later use in the event that hypothesis is found true. A duplicate of the hypothesis pointer is left on the MVP-FORTH parameter stack and is sent along to VERIFY.

VERIFY attempts to perform the proof task. It first checks to see if the pointer left on the stack (at this time, the main hypothesis), is already a "fact on file." VERIFY calls RECALL to see if the pointer represents an already known fact.

If the pointer represents an already known fact, VERIFY exits with the truth value which was returned by RECALL. If the pointer does not represent an already known fact, VERIFY must set about to find some rules that will help support any conclusion.

To find rules, VERIFY sends the pointer along to <FINDRULES. <FINDRULES looks through the rule base for all rules which have, as one of their consequents, the same pointer. Remember that the hypothesis pointer is, in fact, a pointer to a character string. Also, remember that EXPERT-2 allows the same character string to be used as many times as necessary. Thus, if no typographical errors exist in a character string that is used many times, the same pointer will be found in the rule base each time a given string is compiled. Thus, finding rules is the process of looking for the same pointer in the consequent fields of all the rules.

A forward chaining (antecedent reasoning) rule collector, by comparison, also looks for the same pointer. In antecedent reasoning, however, the search takes place in the antecedent fields of all rules, not the consequent fields.

At the beginning of the rule search, a "-1" is left on the MVP-FORTH parameter stack as a flag. This flag tells VERIFY that the last rule has been tested. During the search for rules, when a given rule is found to be applicable, a number is assigned to it, and that number is left on the stack. When all rules have been checked, <FINDRULES returns to VERIFY which now checks to see if the top number on the stack is -1.

If the top number is -1, no rules were found, and VERIFY has no choice but to ask the operator if the hypothesis being tested is true. The answer to that question (yes it is true, or no it is false) becomes VERIFY's response to DIAGNOSE.

If the top number is not -1, VERIFY sets about to prove each of the rules found, looking for a true answer to support the original hypothesis.

Since a typical rule consists of an antecedent field, and a consequent field, and since a typical rule is collected by one of its consequent elements being the same as a hypothesis being tested, then proof of the rule constitutes proof of the hypothesis. Proof of the rule simply means proving each of the antecedent elements. For example, in the following rule:

```
IF animal has feathers
ANDIF animal flies
THEN animal is bird
```

the consequent field element triggered this rule if the hypothesis also happens to be "animal is bird". To prove the animal is a bird, VERIFY must determine if the animal has feathers, and if the animal flies. Each of these statements to prove becomes a new subhypothesis, and VERIFY eventually calls itself to prove them, one at a time.

VERIFY calls itself through one of its subroutines - TESTIF+ - which, itself was called by TRYRULE+, which was called by VERIFY. TESTIF+ calls VERIFY if it needs VERIFY to prove something. TESTIF+ first, however, checks to see if the element it is to prove is a character string, or an analytical subroutine. If the element is an analytical subroutine, that subroutine is executed, and its truth value - after correction for context - is returned to TRYRULE+. If the element is a character string, it is passed to VERIFY by TESTIF+. VERIFY, eventually, will return to TESTIF+ the truth value of the string. That truth value will be corrected for context returned to TRYRULE+.

TRYRULE+ is the teacher in EXPERT-2. Final truth values are returned to TRYRULE+. Any rule found provable has a consequent field that should be saved in the KNOWTRUE stack. Learning, to EXPERT-2, means pushing facts that are deduced through the proof process onto the KNOWTRUE stack. USETHEN, called by TRUERULE+, performs the task of teaching EXPERT-2 what it needs to learn.

Other facts (based on truth values) emerge to EXPERT-2 when VERIFY calls ASK to ask the user if a certain character string is a true statement. The operator returns a true or false indication. ASK calls REMEMBERTRUE if the statement was indicated true by the user. ASK calls REMEMBERFALSE if the statement was indicated false by the user. In either case, the pointer to the particular string is pushed onto one of the facts-on-file stacks. KNOWNTRUE is the stack for all facts determined to be true. KNOWNFALSE is the stack for all facts determined to be false.

A fact may be determined to be true, but that does not make it a proof to EXPERT-2. For example, if "animal is bird" is found to be true, the pointer to that string will be pushed onto the KNOWNTRUE stack. Later in the expert program, the knowledge engineer may have opted to rule out birds as candidates for being a, say, cheetah. This would be accomplished with "IFNOT animal is bird". When this particular antecedent field is encountered by VERIFY, RECALL will find the string to be true. TESTIF+ will get the truth value returned by RECALL and will perform a context check. Since IFNOT places a negative context on the string, false will be returned if the string is true; true will be returned if the string is false.

When VERIFY finds a proof to a given rule, it cleans its stack of any remaining rules by dropping everything on the stack to the -1 flag.

A forward reference is needed for TESTIF+ to call VERIFY. TESTIF+ actually calls an execution vector named VERIFY. MVP-FORTH permits duplicate names. The execution vector consists of a variable ('VERIFY) which is set equal to the code field address of <VERIFY> during compilation of EXPERT-2.

To summarize, the inference machine - DIAGNOSE - uses VERIFY to determine the truth value of a hypothesis. VERIFY collects rules, recalls facts already known, and asks questions of the user. VERIFY converts the antecedent fields of rules it collects into new subhypotheses and sets about to prove each subhypothesis it makes. When a rule proof is found, VERIFY tidies up and returns the final rule truth value to whoever called VERIFY. That, in most cases, is VERIFY. Therefore, VERIFY is a recursive routine that simply ripples through rules looking for a proof. The final proof is returned to DIAGNOSE. DIAGNOSE returns any conclusions that proof might support to the user.

EXPERT-2 is a complete system when compiled on top of an MVP-FORTH language program suitable to the tasks you choose. The expert program you write to execute with EXPERT-2 can be as simple, or as powerful as you choose, depending, of course, on memory size of your computer. However, EXPERT-2 is presented both as a toolkit, and as knowledge engineering teaching tool. It is expected that many EXPERT-2 users will modify portions of EXPERT-2 in order to gain features not presently available. The next section discusses EXPERT-2 modification.

CHAPTER 5

Beyond EXPERT-2

EXPERT-2, as it is released, is primarily a teaching tool. It allows MVP-FORTH programmers to become familiar with expert systems, and, at the same time, create some useful expert programs.

Expert programs can be divided into several categories, according to the purpose, or domain, of the program. Programs that diagnose faults in computers, tractors, or the medical environment are one form of expert domain. Another is interpretation of data, such as, for example, astrometric data taken from a radio telescope, meteorological data taken from an automatic weather station located on a farm, or engine data taken off the diesel engine of a truck tractor during a long-haul trip. Still another domain is teaching or directing the activities of students. Yet another domain is directing the activities of scientific tests in progress, or controlling robots. Natural overlaps occur in these domains. However, selection of a domain generally directs two activities in knowledge engineering:

- o design of the inference engine
- o design of a knowledge representation scheme

EXPERT-2, as it is released, limits the selection of domain for the expert program, unless one is willing to accept the consequences of forcing a fixed representation scheme to operate in a domain in which it is not particuarly efficient.

As a teaching tool for designers of expert systems, EXPERT-2 offers two forms of knowledge representation: analytical equations, and IF-THEN structures. Both are very powerful, especially when combined in scientific work.

EXPERT-2 confines the results of its proof to a binary representation of truth - either TRUE or FALSE. Knowledge engineers know that there are many types of problem domains in which expert systems might operate and for which a binary truth value is inappropriate. Probabilities of proof enter many domains. Weather prediction, for example, typically yields a probability that "it will rain". EXPERT-2 does not provide for the manipulation of probabilities. However, as a shell for clever programmers, probabilities can be added.

Probabilities can be added by exploting unused integer values in the rule compiler. A probability value - say, an integer value between -9 and +9 could be inserted inside parentheses ( ) and the parenthesis symbol defined as a compiler for probability values. Routines can be added to the toolkit to manipulate probabilities using, for example, Bayesian math, or a conditional or fuzzy set math.

Forward chaining (antecedent reasoning) offers knowledge engineers the opportunity to allow the user to volunteer information at the beginning of a session with the expert. With a given character string already known true or false, the inference machine can then set-about to locate any rules that need to know the already-known fact. A candidate hypothesis can be selected from these rules, and the backward chaining system called for a proof. The EXPERT-2 source includes a candidate forward chaining FIND-RULES> which can be called from a different version of the inference machine - DIAGNOSE. To change the way the inference machine works, rewrite DIAGNOSE.

Additional features may be added to EXPERT-2. Some of these features are noted in the source listing; others will emerge as users gain experience.

Perhaps one of the most useful areas to explore enhancement to EXPERT-2 is in the word TELLWHY. When VERIFY finally must ask the user about the truth of a given character string, EXPERT-2 provides the user with the opportunity to find out what the system is up to. ASK permits three possible answers: <Y> for yes the string is true, <N> for no the string is false, and <W> for why did you ask?

When <W> is selected, TELLWHY is called. TELLWHY is a simple version of a teaching tool for the user. TELLWHY lists which hypothesis is being tested, and the current rule VERIFY is dealing with. The entire rule is listed, including any BECAUSE clauses included. Users may wish to enhance this tool by including a listing of the various strings known true and those known false. Users may also wish to add a trace tool to EXPERT-2 which, when enabled, prints select messages added to important routines along the VERIFY path.

Tracing and asking why are two very powerful methods by which a rule base is debugged. In fact, since EXPERT-2 is a logic inference machine, it is very quick to perform any task the rule base directs, even if illogical. Debugging a large rule base is probably one of the more interesting activities a knowledge engineer will have. EXPERT-2 was designed to provide many opportunities to be involved in this activity.

Good luck.

APPENDIX A

BIBLIOGRAPHY

Listed here are documents recommended for readers who need a tutorial on FORTH, and who wish to go beyond the level of information presented in the EXPERT-2 manual.

1- A FORTH Tutorial and Textbook

Starting FORTH — Prentice Hall (Available from Mountain View Press)

By Leo Brodie, formerly of FORTH, Inc. Probably the best book available on the FORTH language. Some minor conflicts will be detected between the FORTH syntax used in Starting FORTH, and the FORTH dialect used for EXPERT-2.

2- An annotated glossary of the FORTH dialect

ALL ABOUT FORTH — Mountain View Press

By Glen B. Haydon. Each entry in the glossary includes a functional definition, an implementation usually in hi-level FORTH, an example of its use and a comment.

3- An Expert Tutorial and Textbook

Building Expert Systems — Addison-Wesley

Edited by Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. At this writing, this is the only text book on expert systems. It reads like a text used in a survey course, and assumes the reader has a background in computer science. The book is enormously useful, and should be studied by all individual who desire to go beyond EXPERT-2.

4- The Animals Game - in LISP

LISP — Addison-Wesley

By P.H. Winston, and B.K.P. Horn. A textbook on the LISP language, with a chapter devoted to a small inference machine - written in LISP -which runs the animal expert game.

5- The Animals Game - in BASIC

Knowledge-Based Expert Systems Come of Age
in Byte Magazine - September, 1981 issue.

By Richard O. Duda and John G. Gaschnig. A good tutorial on expert systems, and a complete discussion of the animals expert game, with a source listing for an inference machine to play the game written in BASIC.

6- Expert Systems Technology Surveys

- Expert Computer Systems
  in IEEE Computer, February 1983 issue
  By Dana S. Nau

- Expert Systems Research
  in Science Magazine, 15 April 1983 issue
  By Richard O. Duda, and Edward H. Shortliffe

- Prospects for Expert Systems in CAD
  in Computer Design Magazine, 21 April, 1983 issue
  By Mark J. Stefik and Johan de Kleer

7- A Look at the Knowledge Engineering Industry

The Fifth Generation Addison-Wesley

By Edward A. Feigenbaum and Pamela McCorduck

Discusses a new breed of supercomputer which is optimized for performing logic inferences.

The bibliography is not intended to be complete. However, most of the references cited contain large, specialized bibliographies from which interested readers can locate references on specific topics related to their needs.

APPENDIX B

USER NOTES

Artificial Intelligence EXPERT systems are composed of an inference machine and a knowledge base. Expert - 2 is such an inference machine. Several small bases of knowledge are included as examples only.

	SCREEN #		TO RUN "TYPE"
EXPERT-2 Source Program Listing	45	LOAD	
Digital Circuit Analyser Rules	90 92	THRU	DIAGNOSE
Animal Classification Program Rules	96 102	THRU	DIAGNOSE
Stock Program and Rules	108 112	THRU	DIAGNOSE
Expert Program Format	117 118	THRU	
Life Automata Rules	119 130	THRU	DIAGNOSE
Weather Prediction Rules	135	LOAD	RUN
Disease Program Rules	155 164	THRU	DIAGNOSE

To Load in the Expert System Boot up SUPERFORTH and type RUN, then from FORTH type: 45 **LOAD** to load the Expert System on top of SUPERFORTH.

Then Load one of the knowledge Base examples listed above(ie. 108 112 THRU for the stock program.) Read instructions on page 18 for operation. To Change the Knowledge base:

```
FORGET WALL
: WALL;
```

and then enter another or your own Knowledge base. Use the demonstration programs as a model. This can be done interactively at the keyboard or you can write your own knowledge base on FORTH screens and then LOAD those screens. A minimal knowledge & understanding of SUPERFORTH will make the use of this program much easier.

```
SCR #45
  0 ( EXPERT-2  LOAD SCREEN                                    MVP-FORTH)
  1 46 48  THRU   (  VARIABLES                                          )
  2 49 50  THRU   (  MISC, SUPPORT                                      )
  3 51 63  THRU   (  VERIFY SUPPORT                                     )
  4 64 66  THRU   (  I/O                                                )
  5 67     LOAD   (  USER SUPPORT                                       )
  6 68     LOAD   (  FACTS SUPPORT                                      )
  7 69 70  THRU   (  More VERIFY SUPPORT                                )
  8 72 73  THRU   (  More VERIFY SUPPORT                                )
  9 74     LOAD   (  INFERENCE MACHINE                                  )
 10 76 79  THRU   (  RULE COMPILER                                      )
 11 81 85  THRU   (  More RULE COMPILER                                 )
 12 ( 90  92    THRU   ( Digital Circuit Analyzer                      )
 13 ( 96  102   THRU   ( Animal classification program                 )
 14 ( 108 112   THRU   ( Stock Program                                 )
 15 ( 117 118   THRU   ( Expert Program Format                         )


SCR #46
  0 ( EXPERT-2  variables, etc 1                               MVP-FORTH)
  1
  2 VARIABLE STRINGSTART      10000 ALLOT
  3    ( This variable marks the beginning of the string buffer.    )
  4    ( It's size may be changed or the buffer moved as desired.   )
  5
  6 VARIABLE RULESTK
  7    ( RULESTK holds pointer to rules.                            )
  8
  9 VARIABLE #RULES
 10    ( #RULES incremented while compiling rules.                  )
 11
 12 VARIABLE SCOUNT
 13    ( SCOUNT is used by the rule compiler.                       )
 14
 15


SCR #47
  0 ( EXPERT-2  variables, etc 2                               MVP-FORTH)
  1
  2 VARIABLE RULE#   40 ALLOT
  3    ( A stack for rules being tested; used by ASK.               )
  4
  5 VARIABLE SUBHYP^
  6    ( The '^' is an ideogram connoting pointer.                  )
  7
  8 VARIABLE CURHYP
  9    ( CURHYP points to current hypothesis.                       )
 10
 11 VARIABLE STRINGS
 12    ( STRINGS has start addr of next string space available.     )
 13
 14 VARIABLE $FLAG
 15    ( For string compiler.                                       )
```

```
SCR #48
  0 ( EXPERT-2  variables, etc 3                              MVP-FORTH)
  1
  2 1 CONSTANT TRUE
  3 0 CONSTANT FALSE
  4
  5 VARIABLE KNOWNTRUE   254 ALLOT
  6    ( True facts will be pushed on the KNOWNTRUE stack.          )
  7
  8 VARIABLE KNOWNFALSE   254 ALLOT
  9    ( False facts will be pushed on the KNOWNFALSE stack.        )
 10
 11 VARIABLE HYPSTACK   254 ALLOT
 12    ( Stack for hypotheses collection during rule compiling.     )
 13
 14 VARIABLE 'VERIFY
 15    ( The variable is used to vector the execution of VERIFY.    )


SCR #49
  0 ( EXPERT-2  misc support 1                                MVP-FORTH)
  1
  2 : REMEMBER?    ( fact, stk base,  --- tf )
  3    BEGIN   DDUP @ DUP 0=   NOT ROT ROT =  NOT   AND
  4    WHILE  2+
  5    REPEAT   SWAP DROP @ ;
  6
  7 EXIT
  8
  9     REMEMBER? checks to see if fact already on
 10     referenced stack.
 11
 12
 13
 14
 15


SCR #50
  0 ( EXPERT-2  misc support 2                                MVP-FORTH)
  1
  2 : PUSH   ( fact^, stk addr --- tf )
  3    DDUP  REMEMBER?    ( Already on STK ? )
  4    IF  DDROP TRUE
  5    ELSE
  6      BEGIN  DUP @
  7      WHILE  2+
  8      REPEAT ! FALSE
  9    THEN ;
 10
 11 FORTH   EXIT
 12
 13     PUSH pushes a fact on a referenced stack. If already
 14     there, returns a true flag. Otherwise, returns a false.
 15
```

```
SCR #51
  0 ( EXPERT-2  verify comments                                MVP-FORTH)
  1 EXIT    This is a comment screen.
  2
  3 Rules are compiled thusly:
  4    Start compiling at "HERE".   Antecedents first
  5 LF <byte> POINTER <word> -antecedent 1
  6 LF <byte> POINTER <word> -antecedent 2
  7 "
  8 LF <byte> POINTER <word> -antecedent N
  9 XX <byte> 00      <word> -consequent end flag
 10 LF <byte> POINTER <word> -consequent 1
 11 LF <byte> POINTER <word> -consequent 2
 12 "
 13 LF <byte> POINTER <word> -consequent N
 14 XX <byte> 00      <word> -consequent end flag
 15   NOTE: XX is unimportant -  available for future changes.


SCR #52
  0 ( EXPERT-2  verify support                                 MVP-FORTH)
  1
  2 : GETLOGIC    ( ruleaddr  ---  ruleaddr, lf  )
  3    DUP 1-  C@  ;
  4
  5    ( GETLOGIC - on entry, TOS ^ any pointer. Leave pointer     )
  6    ( on stack along with associated logic flag < lf >.         )
  7
  8 : FINDTHEN    ( ^1st  if  ---  ^ 1st then )
  9    BEGIN  DUP @
 10    WHILE  3 +
 11    REPEAT  3 +   ;
 12
 13    ( FINDTHEN - On entry, TOS ^ antecedent pointer.  Zip thru  )
 14    ( antecedents.  Leave stack ^ first consequent pointer.     )
 15


SCR #53
  0 ( EXPERT-2  verify support                                 MVP-FORTH)
  1
  2 : STRING?      ( --- )
  3    GETLOGIC  DUP  0=  SWAP   1 =  OR  ;
  4    ( See if logic flag indicates this is a string rule.        )
  5
  6 : RULE#PUSH     ( r#  --- )
  7    RULE#  @ 2+ DUP ROT SWAP ! RULE# ! ;
  8
  9 : RULE#DROP     ( --- )
 10    RULE#  DUP  @ = NOT
 11    IF   -2  RULE#  +!
 12    THEN   ;  EXIT
 13
 14     RULE# words are in support of ASK which needs to know
 15     which rule VERIFY is trying to prove.
```

```
SCR #54
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : <FINDRULES   ( --- -1, r#, r#, ... r# )
  3    -1  ( Leave end flag )   RULESTK @ 1+  ( ^ 1st IF pointer)
  4    #RULES @  0
  5    DO  FINDTHEN
  6      BEGIN DUP @ SUBHYP^ @ @ =
  7        IF  ( found a rule ) I SWAP
  8          FINDTHEN  ( ^ next IF field ) TRUE  ( kill UNTIL )
  9        ELSE  ( not rule yet )  3 + DUP @ NOT
 10          IF  ( out of THENs )  3 + TRUE
 11          ELSE  FALSE      ( try again )
 12          THEN
 13        THEN
 14      UNTIL
 15    LOOP  DROP ;


SCR #55
  0 ( EXPERT-2  verify comments                               MVP-FORTH)
  1 EXIT   This is a comment screen.
  2
  3 <FINDRULES collects rules to test -
  4  leaves stack with :
  5
  6       -1  R#  R# ... R#
  7
  8 If no rules found,  TOS = -1
  9
 10 <FINDRULES  is backward chaining.
 11
 12 FINDRULES>  is forward chaining.
 13
 14
 15


SCR #56
  0 ( EXPERT-2  verify comments                               MVP-FORTH)
  1 EXIT
  2 ( Not implemented  - possible definition )
  3 : FINDRULES>
  4    -1 RULESTK @ 1+
  5    #RULES @ 0
  6    DO
  7      BEGIN DUP @ SUBHYP^ @ @ =
  8        IF I SWAP FINDTHEN TRUE
  9        ELSE 3 + DUP @ NOT
 10          IF 3 + TRUE
 11          ELSE FALSE
 12          THEN
 13        THEN
 14      UNTIL
 15    LOOP DROP ;
```

```
SCR #57
  Ø ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : FINDIF        ( r#  ---  ^ lst antecedent )
  3    RULESTK @ 1+  SWAP ?DUP
  4      IF  (  r# > Ø )  Ø
  5        DO FINDTHEN    (  ^ lst THEN )
  6           FINDTHEN    (  ^ lst IF, next rule )
  7        LOOP
  8      THEN  ;
  9
 1Ø EXIT
 11
 12     FINDIF turns a rule #  ( r# ) into address of first
 13     antecedent pointer field of that rule.
 14
 15


SCR #58
  Ø ( EXPERT-2  verify support                                MVP-FORTH)
  1 : RUNEQN    ( cfa^  --- cfa^, tf  )
  2    GETLOGIC  DUP 2 =  SWAP 3 = OR ( make sure it's a FORTH cfa )
  3      IF ( eqn ) DUP @ EXECUTE  ( --- cfa^, tf )
  4        SWAP GETLOGIC 2 = ROT XOR NOT  ( correct for context )
  5      ELSE ( not eqn ) TRUE  ( try next )
  6      THEN   ;   EXIT
  7 RUNEQN executes FORTH words compiled by IFRUN,
  8  IFNOTRUN, ANDRUN, ANDNOTRUN, THENRUN, and BECAUSERUN.
  9
 1Ø RUNEQN first checks to be sure cfa ^ does now ^ to
 11  BECAUSE ( or other ) string clause.  If not an eqn to
 12  run, leave true for RULECK which follows in TRYRULE+ .
 13
 14 RUNEQN must return to tf.  RUNEQN then corrects the rf
 15  for value of lf ( context ).


SCR #59
  Ø ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : RULECK    ( fact^, tf  --- f, t  or fact^, f )
  3    IF  3 + FALSE   ( fact was true - try next )
  4    ELSE  DROP FALSE TRUE   ( false, exit )
  5    THEN  ;
  6
  7 EXIT
  8
  9 Rules for context correction.
 1Ø    If  eqn-->t, and lf = +, then true
 11    If  eqn-->f, and lf = +. then false
 12    If  eqn-->t, and lf = -, then false
 13    If  eqn-->f, and lf = -, then true
 14
 15
```

```
SCR #60
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : VERIFY        ( hyp^  ---  tf )
  3    'VERIFY  @  EXECUTE ;
  4
  5 EXIT
  6
  7    This is a vectored implementation of VERIFY.  It
  8    allows forward referencing of the actual definition to
  9    the primitive <VERIFY> .
 10
 11    VERIFY is called for string rules - reports truth
 12    TESTIF+ then checks logic flag.
 13
 14
 15
```

```
SCR #61
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : TESTIF+       ( r#  ---  tf )
  3    FINDIF
  4      BEGIN   DUP @ 0=
  5        IF  ( No more IF's, all found true )  DROP TRUE TRUE
  6        ELSE  STRING?
  7          IF   DUP VERIFY    ( IF^, IF^ --- IF^, tf )
  8            SWAP GETLOGIC     ( tf, IF^, lf )
  9            0=               ( true if + logic  )
 10            ROT XOR NOT        ( IF^, tf )
 11          ELSE  ( eqn ) RUNEQN      ( cfa^, tf )
 12          THEN RULECK
 13        THEN
 14      UNTIL   RULE#DROP  ;
 15
```

```
SCR #62
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : XCALL    ( stk^ --- tf )
  3    BEGIN  DUP @ DUP
  4      IF SUBHYP^   @ @ =
  5        IF  DROP TRUE FALSE    ( found )
  6        ELSE  TRUE          ( not found yet )
  7        THEN
  8      ELSE DROP DROP FALSE  FALSE        ( not found anywhere )
  9      THEN
 10    WHILE 2+  ( bump stk^ )
 11    REPEAT    ;     EXIT
 12 XCALL supports RECALL by checking for last entry on
 13  stack ( checks pointers ) equal to current subhyp.
 14
 15 Exits true if found, false if not found.
```

```
SCR #63
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : RECALL    ( fact^ --- tf, t if found )
  3    SUBHYP^ !                    ( save fact as new subhypothesis )
  4    KNOWNTRUE XCALL
  5      IF TRUE    TRUE            ( found true )
  6      ELSE                       ( not found true )
  7        KNOWNFALSE   XCALL
  8          IF  FALSE TRUE        ( found false )
  9          ELSE FALSE            ( not found anywhere )
 10          THEN
 11      THEN    ;
 12
 13
 14
 15


SCR #64
  0 ( EXPERT-2  I/O                                           MVP-FORTH)
  1
  2 : .$      ( string address  --- )
  3    COUNT TYPE ;
  4
  5 : S. CR 2 SPACES .$ ;
  6
  7 : .$$  DUP @ .$ ;
  8
  9 : .NFA   DUP @ NFA ID. ;
 10
 11
 12
 13
 14
 15


SCR #65
  0 ( EXPERT-2  I/O                                           MVP-FORTH)
  1
  2 : T1 CR ." IS THIS TRUE ? (Y=YES, N=NO, W=WHY)" S. ;
  3 : T2 CR ." I CONCLUDE " S.  CR ;
  4 : T3 CR ." I DEDUCE " S.  CR ;
  5 : T4 CR ." CANNOT PROVE ANYTHING " CR ;
  6 : T5 CR ." I AM TRYING TO PROVE" S. ;
  7 : T6 CR ." I AM TESTING RULE# " . ;
  8 : T7 ." IF " 0 ;
  9 : T8 ." IFNOT " 0 ;
 10 : T9 ." IFRUN " 1 ;
 11 : T10 ." IFNOTRUN " 1 ;
 12 : T11 ." BECAUSE " 0 ;
 13 : T12 ." BECAUSERUN " 1 ;
 14 : T13 ." THEN " 0 ;
 15 : T14 ." THENRUN " 1 ;
```

```
SCR #66
  0 ( EXPERT-2  I/0                                          MVP-FORTH)
  1
  2 : CASE:
  3    CREATE  ] SMUDGE
  4    DOES>  SWAP 2*  +  @  EXECUTE ;
  5
  6 : NOP ;
  7    ( Does nothing for some of the cases.                          )
  8
  9 CASE:   ?.  .$$  .NFA ;
 10
 11 CASE:   IF?T  T7 T8 T9 T10 T11 T12 ;
 12
 13 CASE:   THEN?T  T13  NOP T14 NOP T11 T12 ;
 14
 15


SCR #67
  0 ( EXPERT-2  user support                                 MVP-FORTH)
  1
  2 : TELLWHY
  3    CR CURHYP  @ T5 CR
  4    RULE# @ @ DUP T6
  5      FINDIF  CR
  6      BEGIN  GETLOGIC IF?T ?.  CR  3 + DUP @ 0=
  7      UNTIL  3 + ( If's done )
  8    BEGIN GETLOGIC  THEN?T ?.  CR  3 + DUP @ 0=
  9    UNTIL  DROP ;
 10
 11 EXIT
 12
 13 Optional, can add display of KNOWNTRUE and
 14 KNOWNFALSE stacks.
 15


SCR #68
  0 ( EXPERT-2  facts support                                MVP-FORTH)
  1
  2 : REMEMBERTRUE     ( fact^ --- )
  3    KNOWNTRUE PUSH DROP ;
  4
  5 : REMEMBERFALSE    ( fact^ --- )
  6    KNOWNFALSE PUSH DROP ;
  7
  8    ( REMEMBERTRUE & REMEMBERFALSE are methods by which           )
  9    ( EXPERT-2 learns facts.                                      )
 10
 11 : UNLEARN      ( --- )
 12    RULE# DUP !        ( init stack )
 13    KNOWNTRUE 256 ERASE
 14    KNOWNFALSE 256 ERASE   ;
 15    ( Clear all pointers and stacks.                              )
```

```
SCR #69
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : ASK      ( fact^  --- tf  )
  3    BEGIN DUP @  T1    ( ask operator )
  4      KEY ( XKEY ) DUP 2 SPACES EMIT DUP CR
  5      87 ( ASCII  W ) =    ( why did you ask ? )
  6        IF DROP TELLWHY  FALSE
  7        ELSE 89 ( ASCII Y ) = ( is true )
  8          IF  @  REMEMBERTRUE  TRUE
  9          ELSE  @  REMEMBERFALSE FALSE
 10          THEN   TRUE    ( kill UNTIL  )
 11        THEN
 12    UNTIL  ;
 13
 14
 15


SCR #70
  0 ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : USETHEN    ( r# --- )
  3    FINDIF     ( r#  --- ^ first IF  )
  4    FINDTHEN  ( --- ^ first THEN    )
  5    BEGIN DUP @
  6    WHILE DUP  STRING?
  7      IF            ( is string clause )
  8        DUP @ KNOWNTRUE PUSH NOT
  9          IF  DUP @  T3         ( tell )
 10          THEN
 11      ELSE ( execute MVP-FORTH cfa if is analytical sub. <eqn> )
 12        RUNEQN  DROP     ( drop tf  )
 13      THEN   3 +
 14    REPEAT  DROP  ;
 15


SCR #71
  0 ( EXPERT-2  verify comments                               MVP-FORTH)
  1 EXIT
  2
  3 USETHEN pushes all string THENS to KNOWNTRUE,
  4 executes all THENRUN subroutines, and ignores any
  5 BECAUSE clauses or BECAUSERUN's.
  6
  7 On entry, TOS is rule # .
  8
  9 Note: All THENRUN's, like IFRUN's must return a tf.
 10 This can be changed - at some risk - by dropping
 11 DROP after RUNEQN, here.  Pay attention to RUNEQN.
 12
 13
 14
 15
```

```
SCR #72
  Ø ( EXPERT-2  verify support                                MVP-FORTH)
  1
  2 : TRYRULE+       ( r#  --- tf  )
  3    DUP DUP RULE#PUSH      ( save r# for ASK )
  4    TESTIF+                ( now, prove )
  5      IF USETHEN TRUE
  6      ELSE DROP FALSE
  7      THEN  ;
  8
  9
 1Ø
 11
 12
 13
 14
 15


SCR #73
  Ø ( EXPERT-2  verify definition                             MVP-FORTH)
  1 : <VERIFY>         ( hyp^  ---  tf )
  2    DUP RECALL NOT     (  --- tf, t or - f )
  3      IF <FINDRULES DUP -1 =    ( any rules ? )
  4        IF  ( no rules ) DROP ASK    ( --- tf )
  5        ELSE           ( some rules to test )
  6          BEGIN  DUP -1 = ( REMEMBERFALSE fact cannot be proved.)
  7            IF DROP @ REMEMBERFALSE  FALSE TRUE  ( out of rules )
  8            ELSE   TRYRULE+      ( --- tf )
  9              IF  ( found true  )
 1Ø                BEGIN -1 = UNTIL  ( drop rules) DROP TRUE TRUE
 11              ELSE FALSE THEN THEN
 12          UNTIL
 13        THEN
 14      ELSE SWAP DROP
 15      THEN  ;  ( leave truth flag )


SCR #74
  Ø ( EXPERT-2  inference machine                             MVP-FORTH)
  1
  2 ' <VERIFY> CFA 'VERIFY !  ( setup execution vector )
  3
  4 : DIAGNOSE    UNLEARN
  5    HYPSTACK  2-
  6      BEGIN
  7        2+ DUP @ Ø= NOT
  8          IF DUP @ DUP @ CURHYP ! VERIFY ?DUP
  9          ELSE DROP  ( out of hyps to prove )
 1Ø            FALSE TRUE
 11          THEN
 12      UNTIL
 13      IF  ( success ) DROP CURHYP @ T2
 14      ELSE T4
 15      THEN ;
```

```
SCR #75
  0 ( EXPERT-2  inference machine comments                MVP-FORTH)
  1 EXIT
  2   DIAGNOSE is a very simple inference machine.  More
  3    powerful inference machines would combine backward &
  4    forward chaining tools.
  5   HYPSTACK stores pointers in RAM to consequent cells which
  6    have been designated as hypothesis ( By THENHYP ) .  Cells
  7    contain pointers to strings, or CFAs for runtime MVP-FORTH
  8    words.  Hypothesis cells must point to strings because
  9    this expert system collects rules by comparing string
 10    addresses.  THENHYP can only refer to a string consequent.
 11    Other consequents can be strings, or execution words
 12    ( analytical subroutines ) .
 13   VERIFY compares string pointers, or solves equations
 14     which must return a truth flag.
 15


SCR #76
  0 ( EXPERT-2  rule compiler                             MVP-FORTH)
  1
  2 : $=          ( addr1, addr2 --- tf )
  3    TRUE $FLAG !   DUP C@ SCOUNT @ = NOT  ( string counts = ? )
  4      IF DDROP FALSE EXIT THEN
  5    DUP C@ 0   ( this allows 255 byte string check - maximum )
  6      DO  1+ SWAP 1+ OVER C@ OVER C@ = NOT  ( characters = ? )
  7        IF FALSE $FLAG !   LEAVE   THEN
  8     LOOP  DDROP  $FLAG @ ;
  9
 10 EXIT
 11
 12   $= is hi-level string comparison definition.
 13
 14
 15


SCR #77
  0 ( EXPERT-2  rule compiler                             MVP-FORTH)
  1
  2 : FINDCLAUSE    ( addr  ---  f, or addr, t )
  3    STRINGSTART
  4    BEGIN  DUP C@      ( count > 0 ? )
  5      IF DDUP $=
  6        IF  ( found ) SWAP DROP TRUE TRUE
  7        ELSE COUNT + FALSE   ( not yet  )
  8        THEN
  9      ELSE DDROP FALSE TRUE   ( nowhere )
 10      THEN
 11    UNTIL  ;   EXIT
 12   FINDCLAUSE simple checks to see if a string being
 13     compiled from rules already exists in string array which
 14     begins at STRINGSTART.  If it already exists, leave
 15     address where found, and true - else leave false.
```

```
SCR #78
  0 ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : ADD       , ;       ( compile a number into rules                 )
  3 : ADD0    0 C, ;      ( string + logical flag                       )
  4 : ADD1    1 C, ;      ( string - logic flag                         )
  5 : ADD2    2 C, ;      ( eqn    + logic flag                         )
  6 : ADD3    3 C, ;      ( eqn    - logic flag                         )
  7 : ADD4    4 C, ;      ( because string  logic flag                  )
  8 : ADD5    5 C, ;      ( because eqn logic flag                      )
  9
 10
 11
 12
 13
 14
 15


SCR #79
  0 ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : GETCLAUSE      ( --- string addr )
  3    C/L  >IN @ OVER MOD -  BLK @ BLOCK >IN @ +
  4    SWAP DUP >IN +!  -TRAILING  DUP SCOUNT !  ( save count )
  5    DDUP CR TYPE ( type string ) OVER ( addr ) 1- FINDCLAUSE
  6      IF ( found ) SWAP DROP SWAP DROP  ( leave string addr )
  7      ELSE DUP STRINGS @ C!  ( count )
  8        STRINGS @ 1+ SWAP CMOVE  ( move string to string array )
  9        STRINGS @ DUP COUNT + STRINGS !
 10      THEN  ;   EXIT
 11
 12   GETCLAUSE gets rest of line in input stream.  CMOVEs string
 13    to string array.  Leaves string address for compiling
 14    in rule field.
 15


SCR #80
  0 ( EXPERT-2  rules comments                                 MVP-FORTH)
  1 EXIT
  2
  3   Strings are limited to C/L long - including the operator
  4   ( IF, THEN, etc ) .  On Apple II, C/L might = 40, depending
  5   on MVP-FORTH system implementation.  It is possible to
  6   change GETCLAUSE to compile longer strings.
  7
  8   STRINGS holds pointer to byte where tf will be compiled.
  9
 10   -TRAILING leaves  addrress and count.
 11
 12
 13
 14
 15
```

```
SCR #81
  Ø ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 VARIABLE -FINISHED?
  3 VARIABLE -IF-FINISHED?
  4
  5 VOCABULARY RULE IMMEDIATE
  6 RULE DEFINITIONS
  7
  8 : ANDIF    1 -IF-FINISHED? !   ADDØ    ( logic flag )
  9    GETCLAUSE  ADD ;  IMMEDIATE
 1Ø
 11 : ANDIFRUN   1 -IF-FINISHED?  !  ADD2
 12    [COMPILE]  '  CFA ADD ; IMMEDIATE
 13
 14 : AND   [COMPILE] ANDIF  ;  IMMEDIATE
 15


SCR #82
  Ø ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : <IF>   -FINISHED? @
  3    IF ADDØ   Ø  ADD ( end flag for previous rule field   )
  4      Ø -FINISHED? !
  5    THEN  1 #RULES +!  ;
  6   ( If this IF follows a THEN, then compile an end flag        )
  7   ( for the previous consequent field.  If this is the first   )
  8   ( IF  antecedent field, do not compile a flag.               )
  9 : ANDNOT   1 -IF-FINISHED?  ! ADD1  GETCLAUSE ADD ;  IMMEDIATE
 1Ø
 11 : IFNOT   <IF> [COMPILE]  ANDNOT ;  IMMEDIATE
 12
 13 : ANDNOTRUN   1 -IF-FINISHED? !  ADD3 [COMPILE]  '  CFA ADD ;
 14    IMMEDIATE
 15 : IFNOTRUN   <IF> [COMPILE] ANDNOTRUN ; IMMEDIATE


SCR #83
  Ø ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : <THEN>    1  -FINISHED? !  -IF-FINISHED? @
  3    IF ADDØ Ø ADD THEN ;
  4
  5 : IF   <IF>  [COMPILE] ANDIF ; IMMEDIATE
  6
  7 : IFRUN   <IF> [COMPILE] ANDIFRUN ; IMMEDIATE
  8
  9 : ANDTHEN    ADDØ  GETCLAUSE ADD ;  IMMEDIATE
 1Ø
 11 : THEN   <THEN>  [COMPILE] ANDTHEN  ; IMMEDIATE
 12
 13 : ANDTHENRUN   ADD2  [COMPILE]  '  CFA ADD ; IMMEDIATE
 14
 15
```

```
SCR #84
  0 ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : THENRUN    <THEN> [COMPILE] ANDTHENRUN ;  IMMEDIATE
  3
  4 : THENHYP    <THEN> ADD0 HERE GETCLAUSE
  5    ADD HYPSTACK PUSH DROP ;    IMMEDIATE
  6
  7 : ANDRUN    [COMPILE] ANDIFRUN ; IMMEDIATE
  8
  9 : BECAUSERUN    1 -IF-FINISHED? !  ADD5
 10    [COMPILE]  ' CFA ADD ; IMMEDIATE
 11
 12 : BECAUSE     1 -IF-FINISHED? !  ADD4
 13    GETCLAUSE ADD ; IMMEDIATE
 14
 15


SCR #85
  0 ( EXPERT-2  rule compiler                                  MVP-FORTH)
  1
  2 : DONE   ADD0 0 ADD [COMPILE] FORTH DEFINITIONS ;
  3
  4 FORTH DEFINITIONS
  5
  6 : RULES    [COMPILE] RULE  HERE RULESTK ! 0 #RULES !
  7    STRINGSTART  STRINGS !  HYPSTACK 256 ERASE
  8    0 -FINISHED? ! ;
  9
 10  EXIT
 11
 12   RULES starts rule compiling at STRINGSTART.  This is
 13    a memory location which must be selected to allow enough
 14    buffer space to contain the programs rules.
 15


SCR #86
  0 (                                                          MVP-FORTH)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR #90
  0 ( EXPERT-2  Digital circuit analyser                    MVP-FORTH)
  1
  2 : WALL ;      ( something to forget )
  3
  4 RULES     ( start rule compiler )
  5
  6 ( Rule 0                                                          )
  7  IFNOT E is true
  8  ANDNOT F is true
  9  THENHYP Chip 3 is bad
 10
 11 ( Rule 1                                                          )
 12  IF E is true
 13  AND F is true
 14  THENHYP Chip 3 is bad
 15


SCR #91
  0 ( EXPERT-2  Digital circuit analyser                    MVP-FORTH)
  1
  2 ( Rule 2                                                          )
  3  IFNOT C is true
  4  AND E is true
  5  THENHYP Chip 2 is bad
  6
  7 ( Rule 3                                                          )
  8  IFNOT D is true
  9  AND E is true
 10  THENHYP Chip 2 is bad
 11
 12
 13
 14
 15


SCR #92
  0 ( EXPERT-2  Digital circuit analyser                    MVP-FORTH)
  1
  2 ( Rule 4                                                          )
  3  IFNOT E is true
  4  AND C is true
  5  AND D is true
  6  THENHYP Chip 2 is bad
  7
  8 ( Rule 5                                                          )
  9  IF A is true
 10  AND B is true
 11  THENHYP Chip 1 is bad
 12
 13 DONE
 14
 15
```

```
SCR #96
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 : WALL ;      ( something to forget )
  3
  4 RULES     ( start rule compiler )
  5
  6 ( Rule 0                                                        )
  7  IFNOT animal is bird
  8  ANDIF animal has hair
  9  THEN animal is mammal
 10  BECAUSE hairy mild-givers are mammals.
 11
 12 ( Rule 1                                                        )
 13  IFNOT animal is bird
 14  ANDIF animal gives milk
 15  THEN animal is mammal.


SCR #97
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 2                                                        )
  3  IF animal has feathers
  4  THEN animal is bird
  5  BECAUSE is ther a bird with no feathers?
  6
  7 ( Rule 3                                                        )
  8  IF animal flies
  9  AND animal lays eggs
 10  THEN animal is bird
 11
 12 ( Rule 4                                                        )
 13  IFNOT animal is ungulate
 14  AND animal eats meat
 15  THEN animal is carnivore.


SCR #98
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 5                                                        )
  3  IFNOT animal is ungulate
  4  AND animal has pointed teeth
  5  AND animal has claws
  6  AND animal has forward eyes
  7  THEN animal is carnivore.
  8
  9 ( Rule 6                                                        )
 10  IF animal is mammal
 11  AND animal has hoofs
 12  BECAUSE ungulates have hooves
 13  THEN animal is ungulate.
 14
 15
```

```
SCR #99
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 7                                                         )
  3  IF animal is mammal
  4  AND animal chews
  5  BECAUSE ungulates chew cud
  6  THEN animal is ungulate
  7  ANDTHEN animal is even toed.
  8
  9 ( Rule 8                                                         )
 10  IF animal is mammal
 11  AND animal is carnivore
 12  AND animal has tawny color
 13  AND animal has dark spots
 14  THENHYP animal is cheetah.
 15
```

```
SCR #100
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 9                                                         )
  3  IF animal is mammal
  4  AND animal is carnivore
  5  AND animal has tawny color
  6  AND animal has black stripes
  7  THENHYP animal is tiger.
  8
  9 ( Rule 10                                                        )
 10   IF animal is ungulate
 11   AND animal has long neck
 12   AND animal has long legs
 13   AND animal has dark spots
 14   THENHYP animal is giraffe.
 15
```

```
SCR #101
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 11                                                        )
  3  IF animal is ungulate
  4  AND animal has black stripes
  5  THENHYP animal is zebra.
  6
  7 ( Rule 12                                                        )
  8  IF animal is bird
  9  ANDNOT animal flies
 10  ANDNOT animal swims
 11  AND animal has long neck
 12  AND animal is black and white
 13  THENHYP animal is ostrich.
 14
 15
```

```
SCR #102
  0 ( EXPERT-2  animal classification program              MVP-FORTH)
  1
  2 ( Rule 13                                                        )
  3  IF animal is bird
  4  ANDNOT animal flies
  5  AND animal swims
  6  BECAUSE penguin is bird that swims
  7  AND animal is black and white
  8  THENHYP animal is penguin.
  9
 10 ( Rule 14                                                        )
 11  IF animal is bird
 12  AND animal flies
 13  AND animal flies well
 14  THENHYP animal is albatros.
 15 DONE


SCR #103
  0 (                                                      MVP-FORTH)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15


SCR #104
  0 (                                                      MVP-FORTH)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR #108
  0 ( EXPERT-2  Stock Program                                MVP-FORTH)
  1
  2 : INPUT#  QUERY BL WORD NUMBER  ;
  3
  4 VARIABLE CB
  5
  6 : ZERO-CB  0 CB ! ;
  7
  8 : CB?  CB @ 0> ;
  9
 10 : BULL?   CB @ 100 > ;
 11
 12 : BEAR?   CB @ -100 < ;
 13
 14
 15


SCR #109
  0 ( EXPERT-2  Stock Program                                MVP-FORTH)
  1
  2 : START
  3    PAGE CR ." New data ? < Y/N >  " KEY
  4    CR CR 89 ( Y ) =
  5    IF CR ." Input stock advances: "  INPUT#
  6       CR ." Input stock declines: "  INPUT# D-
  7       CR ." Input stocks traded:  " INPUT#
  8       DROP 100 SWAP M*/ DROP  CB +!
  9       CR CR ." The new market strength = "   CB @ .
 10       CR CR
 11    THEN  1 ;
 12
 13
 14
 15


SCR #110
  0 ( EXPERT-2  Stock Program                                MVP-FORTH)
  1
  2 : WALL ;
  3
  4 RULES
  5
  6 IFRUN START
  7 THEN data available
  8
  9 IF data available
 10 ANDRUN CB?
 11 AND today's stock quotes are above yesterday's
 12 THEN market is bullish
 13
 14
 15
```

```
SCR #111
  Ø ( EXPERT-2  Stock Program                                  MVP-FORTH)
  1
  2 IF market is bullish
  3 ANDRUN BULL?
  4 THEN market is strongly bullish
  5
  6 IF market is bullish
  7 ANDNOTRUN BULL?
  8 AND today's trading volume is more than twice normal
  9 THEN bull run is over
 1Ø
 11 IF data available
 12 ANDNOT market is bullish
 13 ANDRUN BEAR?
 14 THEN market is strongly bearish
 15


SCR #112
  Ø ( EXPERT-2  Stock Program                                  MVP-FORTH)
  1 IF data available
  2 ANDNOT market is bullish
  3 ANDNOTRUN BEAR?
  4 AND today's trading volume is more than twice normal
  5 THEN bear run is over
  6
  7 IF bear run is over
  8 THENHYP indications are to buy
  9 IF bull run is over
 1Ø THENHYP indications are to sell
 11
 12 IFNOT bear run is over
 13 ANDNOT bull run is over
 14 THENHYP do nothing -- wait
 15 DONE


SCR #113
  Ø (                                                          MVP-FORTH)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR #117
  Ø ( EXPERT-2  Expert Program format                    MVP-FORTH)
  1
  2 : WALL ;
  3
  4 : DEMOWORD
  5    CR ." ENTER A NUMBER ( 1 TO 1Ø ) "
  6    CR KEY 53 >
  7    IF TRUE
  8    ELSE FALSE
  9    THEN ;
 1Ø
 11
 12
 13
 14
 15
```

```
SCR #118
  Ø ( EXPERT-2  Expert Program format                    MVP-FORTH)
  1
  2 RULES
  3
  4 ( RULE Ø )
  5   IF we are ready to begin
  6   ANDIFRUN DEMOWORD
  7   BECAUSE a number > 5 is needed
  8   THEN a number > 5 is available
  9
 1Ø ( RULE 1 )
 11   IF a number > 5 is available
 12   THENHYP all conditions are satisfied
 13
 14 DONE
 15
```

```
SCR #119
  Ø (                                                    MVP-FORTH)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

ARTIFICIAL INTELLIGENCE PACKAGE EXPERT SYSTEM TUTORIAL

for SUPER-FORTH 64™

By Mitch & Linda Derick

The Expert-2 Tutorial is designed to be used in conjunction with the SUPER-FORTH Expert System Toolkit

documentation by Jack Park.

TABLE OF CONTENTS

LIST OF FIGURES

EXPERT-2 TUTORIAL

Introduction

If you want to use the Expert-2 system but are not familiar with computers and/or FORTH language programming, then this tutorial is for you. In this section, we shall answer the four most important questions a beginning user needs to know:

1. What does Expert-2 do for me?

2. How do I write an Expert-2 program?

3. How can I write simple FORTH routines?

What Does Expert-2 Do For Me?

In essence, Expert-2 is a way of simulating the knowldge and experience an expert in a particular field of learning might have.

You, or another expert, formulate one or more hypothesis that you wish to prove. You go about proving each hypotheses by asking the user a series of questions to which he usually answers "yes" or "no." As in "Animal, Vegetable or Mineral," careful selection of questions eliminates entire categories of possibilities and allows the user to come to the correct conclusion.

In the example we will implement later, you are an "expert" on childhood diseases. Your goal is to identify one of seven specific diseases by asking questions about the symptoms. You will form seven hypothesis (one for each disease) and then try to prove one of these to be correct. This is accomplished by both eliminating diseases for which the symptoms do not match and by proving one hypotheses to be true because the symptoms do match.

This process of asking questions, discarding hypothesis that do not fit the given answers and concluding that one specific hypotheses is true is what Expert-2 can do for you.

How Do I Write an Expert-2 Program?

In order to write an Expert-2 program, we must first understand in general terms what the Expert-2 system is and how it works.

Expert systems fall under a category of computer science called "artificial intelligence." With today's state of the art, that means that an expert system is about as intelligent as saccharin is sugar. Saccharin may taste sweet, it may work fine for lots of things--but it just is not sugar. Likewise an expert system can very capably do the job it is intended to do but any intelligence it possesses has been put there by you, the expert. Even then, Expert-2 is not so much artificially intelligent as it is an "automated classifier."

By asking questions in an intelligent order, you can guide the user through a series of computer-aided classifications. This is very similar to the science of taxonomy: the science of classifying animals and plants into hierarchies of superior and subordinate groups based upon their characteristics.

So how does Expert-2 go about "classifying" questions and answers until it can logically come to a conclusion, i.e., prove a hypotheses? Let us make a high level "first pass" to get the general overall picture; and then examine things in more detail.

The primary point to keep in mind is that the GOAL of Expert-2 is to conclude, or prove, a particular hypotheses to be true. To accomplish this, Expert-2 does the following:

1. Uses QUESTIONS and USER-ROUTINES to establish FACTS.

2. Uses FACTS to make DEDUCTIONS.

3. Uses both FACTS and DEDUCTIONS to prove HYPOTHESIS.

Since we are dealing with a computer system, we must have a formal method of describing these QUESTIONS, USER-ROUTINES, FACTS, DEDUCTIONS and HYPOTHESIS that is understandable both by us and by the computer. This problem is cleverly solved by the use of RULES.

The RULE

A RULE is the basic building block of the Expert-2 system. The entire purpose of a RULE is to establish a FACT or FACTS and then make a DEDUCTION or HYPOTHESES based upon these facts. An Expert-2 program is made up of a collection or set of rules. These rules are LOADed into the computer and processed by Expert-2 into an Expert-2 program.

Expert-2 keeps special track of those RULES that conclude with a HYPOTHESES (a THENHYP statement). These become the goal that Expert-2

tries to reach. Expert-2 will attempt to prove each HYPOTHESES true in the order it compiled them. If the "current" HYPOTHESES is proven to be not true, i.e., a QUESTION in its question section is answered "false"; this RULE is discarded. Then the next HYPOTHESES in line is made "current" and the system begins to prove it true or false.

Once any HYPOTHESES is proven true, Expert-2 types its conclusion on the console and stops. If all HYPOTHESIS are discarded (proven false), Expert-2 then types "CANNOT PROVE ANYTHING" on the console and stops.

Just what is a RULE? A RULE is the mechanism Expert-2 uses to convert FACTS into QUESTIONS, and QUESTIONS into DEDUCTIONS and HYPOTHESIS. A RULE is composed of two sections: a question section and a conclusion section. Basically you, the expert, ask the user questions in the question section, and then, based on the users answers to those questions, you form a DEDUCTION or HYPOTHESES in the conclusion section. The basic format of a RULE is:

```
                operator  QUESTION-statement
                                                    Question Section
  (optional)    operator  QUESTION-statement

                operator  DEDUCTION or HYPOTHESES
                                                    Conclusion Section
  (optional)    operator  DEDUCTION or HYPOTHESES
```

You may ask any number of QUESTIONS within a RULE. The user must answer "yes" or "no" to each QUESTION. Once a QUESTION is answered it becomes a FACT. Expert-2 keeps track of these answers/facts for you. If all of the QUESTIONS you asked in the question section are answered with a "yes" then Expert-2 will mark all of the DEDUCTIONS or HYPOTHESIS in the conclusion section to be true. If one or more QUESTIONS are answered with a "no," then the conclusion section will be marked false.

When executing, Expert-2 asks your questions by simply copying the "QUESTION-statement" onto the CRT. For example, the QUESTION statement:

```
IF  animal has hair
```

would be displayed as:

```
IS THIS TRUE ? (Y=YES, N=NO, W=WHY)
  animal has hair
```

The user would respond with a "Y" or "N." (We will talk about "W"

later) thereby converting the QUESTION-statement "animal has hair" into a remembered FACT: either "animal does have hair" OR "animal does not have hair."

Once all FACTS have been established, the DEDUCTION/HYPOTHESES portion of the RULE can then make a statement. For example:

THEN animal is mammal

makes either the DEDUCTION that "animal is a mammal" if the question section was all "true" or the DEDUCTION is inferred that the "animal is not mammal" if part of the question section was "false."

Another example is:

THENHYP animal is mammal

would make a HYPOTHESES that the "animal is mammal." Note that the negative form of a HYPOTHESES cannot be referred to in other RULES.

By now you are probably wondering how the system "knows" what QUESTIONS have been asked, what FACTS have been resolved and what DEDUCTIONS have been made. In other words, how are all of the RULES "glued" together into one logical entity? The key to this answer lies in the "statement" portion of the RULES.

A statement is simply a string of characters that follows an operator and states a FACT. The way it glues RULES together is utter simplicity! While LOADing, Expert-2 simply performs a character by character comparison on each and every statement. Those statements that are identical in every way are then logically linked or "glued" together by Expert-2.

Note the phrase, "identical in every way." Expert-2 is dumb. To a human, the statement "animal is mammal" is the same as the statement "Animal is mammal." To Expert-2 the capitalized "A" in "Animal" is totally different than the lowercase "a" in "animal." These two statements would be considered different and would not be logically linked together by Expert-2. Likewise, typographical errors would cause statements not to be linked. When testing an Expert-2 program, watch for questions being asked more than once. If you notice any, check your screens carefully for a typo.

Let us see what all of this means by examining some rules similar to a portion of the animal program listed in the manual. (Refer to Figure T-1.)

```
( RULE 1                                                    )
IF animal flies
AND animal lays eggs
THEN animal is bird


( RULE 2                                                    )
IFNOT animal is bird
ANDIF animal gives milk
THEN animal is mammal

( RULE 3                                                    )
IF animal is mammal
AND animal has hoofs
BECAUSE ungulates have hoofs
THEN animal in ungulate
```

Figure T-1: EXAMPLE RULES

First look at RULE 1. When RULE 1 executes, two QUESTIONS will be asked; does the animal fly and does it lay eggs? When the QUESTIONS are answered, two FACTS will be known by the system: whether the animal does or does not fly and whether the animal does or does not lay eggs. No matter what the response, something will have also been learned about the DEDUCTION "animal is bird." Either it is a bird or it is not. The crux, the very heart of how Expert-2 works is what happens next. Every single RULE that contains the identical statement "animal is bird" (follow the arrows) now automatically "knows" what the result of RULE 1's DEDUCTION was!

Another example is the "animal is mammal" DEDUCTION in RULE 2. Once RULE 2 is "satisfied," then all other RULES containing "animal is mammal" know the answer.

Now that we know somewhat how Expert-2 works, let us take a look at the tools (operators) we have available with which to write RULES.

There are three basic operators: IF, AND and THEN. Each of which can be modified by additional suffix words. Let us examine the three basic words using RULE 1 as our example:

Basic Operators: IF, AND and THEN

There are two operators that ask QUESTIONS in the question section of a RULE: IF and AND.

IF

IF is used in the form:

IF statement

where the statement following IF is used to ask a QUESTION. (The "statement" portion of an IF phrase is referred to as an <antecedent character string> in the EXPERT-2 manual.)

An example of IF is:

IF animal flies

In this example, the statement "animal flies" will be typed onto the CRT by Expert-2 and the user will give a true or false ("Y" or "N") reply.

Note that the context of IF as used within a RULE is identical to the common English usage of "if."

A RULE must always begin with some form of an IF-type operator. IF statements return a "truth flag" to Expert-2 which reflect the user's answer to the QUESTION asked.

AND

AND is used in the form:

AND statement

where the statement following AND is used to ask an additional question. (The statement portion of an AND phrase is referred to as an <antecedent character string> in the manual.)

An example of AND is:

IF animal flies
AND animal lays eggs

In this example, the "animal lays eggs" will be typed onto the CRT by Expert-2 and the user will give a true or false reply ("Y" or "N").

The purpose of AND is to allow additional QUESTIONS, i.e., FACTS, to be gathered within a single rule to support making a DEDUCTION or proving a HYPOTHESES. This usage is identical to the English usage of "and," i.e., "If this and that and those ...."

The use of AND within a RULE is optional. AND statements also return a "truth flag" to Expert-2 which reflect the user's answer to the QUESTION asked.

THEN-type operators are used in the conclusion section of a RULE.

THEN

THEN is used in the form:

THEN deduction-statement

where the deduction-statement following THEN is used to make a DEDUCTION. Note: A special form of THEN (THENHYP) is used to make a HYPOTHESES. (The statement portion of a THEN phrase is referred to as a <consequential character string> in the manual.)

An example of THEN is:

THEN animal is bird

In this example the statement "animal is bird" will be made into a DEDUCTION (either true or false) when both QUESTIONS of RULE 1, Figure T-1, have been turned into FACTS.

The purpose of THEN is to conclude a sequence of IF and AND operator phrases. The usage is identical to the English usage of "then," i.e., "if this and if that is true (or false) then such and such is also true (or false)."

Some form of the THEN operator (THEN or ANDTHEN) must always be used to terminate an IF/AND sequence within a RULE. THEN statements expect to be supplied with at least one "truth flag" from previous IF or IF/AND statements.

Negative Context Operator Suffixes

Just as it is possible in English to ask a question using "positive logic," e.g., "If the animal flies ..."; it is also possible to ask a question using "negative logic," e.g., "If the animal does not fly ...." This is the purpose of the NOT operator suffix. There are two forms of the NOT suffix; IFNOT and ANDNOT. Let us examine them more closely again using Figure T-1 as an example.

IFNOT

IFNOT is used in the form:

IFNOT statement

where the statement following the IFNOT is used to establish the opposite of a known fact. (The statement portion of an IFNOT phrase is referred to as an <antecedent character string> in the manual.)

An example of IFNOT is:

IFNOT animal is bird

If the statement is established as a FACT by an IF or AND phrase, or the statement is established as a DEDUCTION by a THEN

phrase; then all IFNOTs or ANDNOTs containing the same statement are set to the opposite "truth value."

As a general rule, IFNOTs or ANDNOTs should be used in place of asking the user a "negative-logic" QUESTION. Rather than asking the user a negative-logic QUESTION like "animal does not have feathers"; instead ask a positive-logic QUESTION like "animal has feathers" and reverse the logic in another RULE via a NOT suffix. Otherwise your Expert-2 program will be quite confused with double negatives like: IFNOT "animal does not have feathers."

ANDNOT

ANDNOT is used in the form:

ANDNOT statement

where the statement following the ANDNOT is used to establish the opposite of additional known FACTS. (The statement portion of an ANDNOT phrase is referred to as an <antecedent character string> in the manual.)

An example of ANDNOT is:

ANDNOT animal has feathers

Note that the ANDNOT operator only establishes the opposite of additional known facts (just as AND acts). Like IFNOT, it does not ask a QUESTION. If a RULE containing our example statement "animal has feathers" is executed, the statement will not be typed onto the CRT by Expert-2.

However, if the statement is established as a FACT by an IF or AND phrase, or the statement is established as a DEDUCTION by a THEN phrase; then all IFNOTs or ANDNOTs containing the same statement are set to the opposite "truth value."

As a general rule, IFNOTs or ANDNOTs should be used in place of asking the user a "negative-logic" QUESTION. Rather than asking the user a negative-logic QUESTION like "animal does not have feathers"; instead ask a positive-logic QUESTION like "animal has feathers" and reverse the logic in another RULE via a NOT suffix. Otherwise your Expert-2 program will be quite confused with double negatives like: IFNOT "animal does not have feathers."

Expanding the Power of Expert-2 with Subroutines

Subroutines are a way of greatly increasing the power and flexibility of Expert-2. By using subroutines, you can allow Expert-2 to input numerical values, directly control its own inputs, print large explanations, input complex answers or anything else you can imagine. The only catch is that you must program these subroutines yourself using FORTH.

Since Expert-2 is written in FORTH, all of the features of FORTH become available to Expert-2. This includes any subroutines written to be used by Expert-2.

All subroutines that you would want to use must be LOADed before the word RULES. We will talk in detail about writing simple subroutines in the section on "Programming in SUPER-FORTH."

Just what is a subroutine? A subroutine is a stand-alone, special purpose routine that is written by you in FORTH to perform a specific function that cannot be done for you by Expert-2. It is a small, custom-made "mini program" that is "called" by Expert-2, executes and does its special job, and "returns" back to Expert-2 to the next phrase in the RULE.

Subroutines are "called" from Expert-2 via the suffix word RUN. This usage is the same as the English use of the word "run," i.e., go run subroutine XYZ.

The general format of a RUN-type operator is:

```
operator  subroutine-name
```

where subroutine-name is the name of a previously defined subroutine.

Now let us examine each RUN-type operator in detail.

RUN-Type Operators

IFRUN

IFRUN is used in the form:

```
IFRUN  subroutine-name
```

where the subroutine-name following IFRUN is called when the operator executes. Note that an IFRUN phrase does not contain a "statement." This is because the operator does not ask a QUESTION on the CRT to establish a FACT. Instead, the called subroutine must return a "truth value" which will be used in determining the truth of the DEDUCTION or HYPOTHESES. (More about this "truth value" in the section on programming in FORTH.)

When a RULE executes, any IFRUN phrase within the RULE will always execute and call its associated subroutine.

An example of IFRUN is:

```
IFRUN  INPUT.TEMPERATURE
```

In this example, the subroutine "INPUT.TEMPERATURE" will be called whenever the RULE containing it executes.

ANDRUN

ANDRUN is used in the form:

```
ANDRUN  subroutine-name
```

where the subroutine-name following ANDRUN is called when the operator executes. Note that an ANDRUN phrase does not contain a "statement." This is because the operator does not ask a QUESTION on the CRT to establish a FACT. Instead, the called subroutine must return a "truth value" which will be used in determining the truth of the DEDUCTION or HYPOTHESES. (More about this "truth value" in the section on programming in FORTH.)

An example of ANDRUN is:

```
ANDRUN  INPUT.TIME
```

In this example, the subroutine "INPUT.TIME" will be called whenever the RULE containing it executes.

The purpose of ANDRUN is to allow a subroutine or subroutines to be called within a RULE in addition to whatever form of IF operator is used.

The use of ANDRUN within a RULE is optional.

THENRUN

THENRUN is used in the form:

```
THENRUN  subroutine-name
```

where the subroutine-name following THENRUN is called when the operator executes. Note that a THENRUN does not form a conclusion. It simply calls a subroutine.

This is an important point to keep in mind: A RULE must come to a conclusion. It does not matter whether a DEDUCTION or a HYPOTHESES is proven true or false. The important point is that the RULE must have at least a THEN or a THENHYP. A THENRUN may be added only in addition to one of these two operator types.

A THENRUN or ANDTHENRUN will only execute when all IF or AND-type phrases in a RULE have been satisfied as being true.

An example of THENRUN is:

```
THEN  deduction-statement
THENRUN  EXPOUND.CONCLUSION
```

In this example, the subroutine EXPOUND.CONCLUSION will execute

when the previous THEN has been satisfied.

ANDTHENRUN

ANDTHENRUN is used in the form:

```
ANDTHENRUN  subroutine-name
```

where the subroutine-name following ANDTHENRUN is called when the operator executes. This operator is similar to ANDTHEN in that it allows an additional subroutine to be called whenever a THENRUN executes.

An ANDTHENRUN must be preceded by a THENRUN; it cannot "stand alone." However, multiple ANDTHENRUN's may be included in the conclusion portion of a RULE.

An example of ANDTHENRUN is:

```
THEN  deduction-statement
THENRUN  first.subroutine
ANDTHENRUN  second.subroutine
```

In the example, the subroutine "second.subroutine" will execute after the previous THEN has been satisfied and the previous THENRUN subroutine "first.subroutine" has executed.

THENHYP-Type Operator

THENHYP

THENHYP is used in the form:

```
THENHYP  hypotheses-statement
```

where the statement following THENHYP is used to make a HYPOTHESES. (The statement portion of a THENHYP is referred to as a <consequent character string> in the manual.)

An example of THENHYP is:

```
THENHYP  animal is yellow bellied sapsucker
```

In this example, the statement "animal is yellow bellied sapsucker" will be made into a HYPOTHESES if all IF and AND-type phrases in the RULE are true. Otherwise the HYPOTHESES will be proven false and discarded.

BECAUSE-Type Operators

The purpose of BECAUSE-type operators is to give you (the expert) a way to explain your reasoning to the user. When writing a BECAUSE-type phrase, you should put yourself in the user's place. The series of QUESTIONS you were leading the user through prompted him to answer with a "why?"; instead of "yes" or "no." By putting yourself in the users place, you should be able to see why he asked for further assistance and compose an explanation to answer this question. BECAUSE-type phrases can be placed anywhere desired within a RULE.

BECAUSE-type operators execute only when the user responds to a QUESTION with a "W" ("Why?"). After BECAUSE-type operators execute, Expert-2 re-asks the QUESTION it had originally asked. There are two types of BECAUSE phrases; BECAUSE and BECAUSERUN.

BECAUSE

BECAUSE is used in the form:

```
BECAUSE  explanation
```

where the explanation can be as many characters as will fit on the remainder of the line (64 characters total). This explanation will be printed out along with the rest of the RULE whenever the user responds to a QUESTION with the answer "W" ("Why?").

An example of BECAUSE is:

```
BECAUSE  INSECTS lay eggs and fly too
```

This whole line would be typed onto the CRT by Expert-2 and would hopefully explain to the user what it was your line of reasoning was for the QUESTION Expert-2 had just asked.

BECAUSERUN

BECAUSERUN is used in the form:

```
BECAUSERUN  subroutine-name
```

where the subroutine-name following BECAUSERUN is called when the operator executes. i.e., Whenever the user types in a "W" ("Why?") in response to a QUESTION.

An example of BECAUSERUN is:

```
BECAUSERUN  ?INSECTS-ARE-NOT-BIRDS
```

The subroutine "?INSECTS-ARE-NOT-BIRDS" would execute and do whatever you programmed it to do; type out a detailed explanation, type out an explanation plus data the system had collected via other RUN-type operators, etc.

Think of BECAUSERUN as a more "intelligent" form of BECAUSE whose purpose is still to clarify your line of thinking to the user.

Miscellaneous Operators:

There are several operators that serve no purpose other than to "tell" Expert-2 what your intentions are. These operators will now be discussed.

WALL

WALL is used in the form:

```
: WALL ;
```

where the colon, the characters "WALL" and the semicolon are all considered to be part of WALL. Note the colon and semicolon each must be separated from "WALL" by at least one blank or space. (WALL is actually being defined as a FORTH word that does nothing but serve as a placeholder. Refer to the section on Programming in FORTH for the reasoning behind the : and ;.)

WALL is used to "flag" the beginning of your Expert-2 program (both subroutines and RULES). WALL should be the very first word in your program since it serves as a boundary or "wall" between your program and the SUPER-FORTH/Expert-2 code.

WALL is used in conjunction with the FORTH word FORGET. Typing FORGET WALL (cr) will cause the system to "forget" everything loaded into the system up to and including WALL.

RULES

RULES is used in the form:

```
RULES
```

RULES is a word that tells Expert-2 to get ready to accept your set of RULES. RULES should be placed after any subroutines you created but before any RULES.

What RULES does is to tell SUPER-FORTH to treat any further text loaded into the system as Expert-2 RULES and not SUPER-FORTH instructions.

NOTE: An error message of "CONDITIONALS NOT PAIRED" when loading RULES is a good sign that you possibly forgot to include "RULES" in your program code.

DONE

DONE is used in the form:

DONE

DONE is basically the opposite of the word "RULES." DONE should be the last word in your program.

What DONE does is tell Expert-2 that no more RULES will be loaded, so do any final processing that is necessary and process any further text input as SUPER-FORTH instructions.

DIAGNOSE

DIAGNOSE is used in the form:

DIAGNOSE (cr)

where DIAGNOSE (cr) is typed in from the console.

DIAGNOSE, as its name implies, is the word that tells EXPERT-2 to begin "diagnosing" a user's problem.

Writing an Expert-2 Program

Now that we have covered the basics of Expert-2, let us use this knowledge to create a simple Expert-2 program.

There are many ways of writing a program. What we want to show you is how to write your thoughts in an organized manner, so that the program is understandable to Expert-2 and will lead the user logically toward a definite goal or hypotheses.

If you have never done any programming before, the most obvious approach is to simply sit down and start writing or "coding" Expert-2 RULES. Unfortunately, this simplistic approach to problem solving will almost always end up in frustration and disaster. Fortunately, though, over the years the computer science world has developed simple guidelines that (when followed) assure success almost as surely as the sit-down-and-code approach assures disaster.

These guidelines can be applied to Expert-2 in six straight-forward steps:

1. Formally state your goal/goals, i.e., the eventual HYPOTHESES/HYPOTHESIS you wish to prove.

2. List the FACT/FACTS needed to reach the goal/goals.

3. Put these FACTS into chart form.

4. Classify these FACTS into general categories of "natural groupings."

5. Develop your logic flow using the "divide and conquer" approach.

6. Convert the FACTS into QUESTIONS, i.e., RULES.

GUIDELINE 1: Formally state your goal/goals, i.e., the eventual HYPOTHESES/HYPOTHESIS you wish to prove.

Do not let the simplicity of this step lull you into skipping over it. Countless billions of dollars have been wasted in the programming industry because of programmers who omitted this step. You have to know where you are going. The whole purpose of Expert-2 is to use your knowledge to lead a user from Point A to Point B. You cannot lead the user to Point B if you yourself are unsure of what or where Point B is. Or, even worse, you cannot get to Point B if your concept of Point B keeps evolving as you code.

One point to keep in mind is that, if you cannot state your goal in a succinct, concise manner, there is no way you are going to be able to program it successfully. The "I'll start coding now and things will become more obvious as I go along" attitude only makes it obvious that you should not have skipped this first step!

OK. So let us formally state our problem in as much detail as necessary to explicitly cover all of the bases.

GOAL: By asking questions about a subject's symptoms, identify one of seven childhood diseases.

Develop a set of Expert-2 RULES that will ask QUESTIONS concerning symptoms and will arrive at one of seven possible HYPOTHESIS.

The seven childhood diseases that we wish to either prove as a true HYPOTHESES or discard as a false HYPOTHESES are:

Measles
Mumps
Chicken Pox
German Measles
Flu
Whooping Cough
Common Cold

That pretty well sums up what our sample program is going to do.

GUIDELINE 2: List the FACT/FACTS needed to reach the goal/goals.

Just as Guideline 1 makes clear what our goal is; Guideline 2 tells us exactly what we need to prove to reach it.

Here we list the information needed to develop our QUESTION/RULE base. We must supply enough information to be able either to positively prove or disprove each HYPOTHESES. On the other hand, too much information can lead to user confusion and possibly wrong conclusions. Fortunately, you are the expert in this realm of knowledge and the correct choices should readily become apparent. This information must be stated such that it can be phrased as an Expert-2 QUESTION.

The following are the facts needed to be ascertained for our Childhood Diseases program. These facts are summarized from SYMPTOMS The Complete Home Medical Encyclopedia, edited by Sigmund S. Miller, published by Avon Books, NY, NY.

F A C T S

HYPOTHESES 1: Measles

Symptoms:
- Runny nose
- High temperature -- fast rising
- Harsh, hacking cough
- Bloodshot eyes
- Conjunctivitis
- White spots inside cheeks
- Heavy, brownish-pink rash first on scalp and neck, then body

HYPOTHESES 2: Mumps

Symptoms:
- Swelling of salivary glands
- Moderate to high temperature
- Painful or impossible to suck on lemon
- Scant or excessive saliva
- Lymph node in neck enlarged

HYPOTHESES 3: Chicken Pox

Symptoms:
- Spotty rash, pimples, blisters all at same time
- Achy body
- High temperature
- Chills

HYPOTHESES 4: German Measles

Symptoms:
- Runny nose
- Slight temperature
- Light, rose-colored rash first on face and neck, then body
- Lymph nodes behind ear enlarged
- Stiff joints
- Headache

HYPOTHESES 5: Flu

Symptoms: Abrupt temperature and chills
Severe headache
Strong body aches
Weakness
Runny nose
Sore throat
Inflamed mucus membrane of eye
Cough
Vomiting
Diarrhea

HYPOTHESES 6: Whooping Cough

Symptoms: Runny nose
Sneezing
Tearing
Light, hacking cough (at first)
Loss of appetite
No temperature
Dozen coughs, then whooping air intake

HYPOTHESES 7: Common Cold

Symptoms: Moderate temperature
Sore throat
Runny nose
Sneezing
Mild body aches
Moderate headache

As you can see, listing the facts for each HYPOTHESES is not a trivial matter. Documenting a large complex problem can be very time consuming and tedious. You should be aware, however, that any time spent formally documenting the facts will be less than the time spent by skipping this step and attempting to get an undocumented program debugged.

This listing or documenting of the facts has four important purposes.

1. All of the facts needed to reach a specific HYPOTHESES are collected in <u>one</u> place rather than spread out in an Expert-2 program's RULE set.

   This allows you to see both missing facts needed to reach the goal and additional facts that could confuse reaching the goal.

2. Exhaustively writing each fact necessary to reach a goal reveals <u>commonality</u> between HYPOTHESIS. This

allows you to more easily perform Guideline 3 (Break the facts into natural groupings). If a certain fact or group of facts are necessary to prove two or more HYPOTHESIS, they may be grouped together to form a DEDUCTION (via a THEN-type statement) and thereby greatly reducing the number of IF and AND-type QUESTION statements you must program in the RULE set.

3. Having all of the facts for each HYPOTHESES in one place is invaluable when you start debugging your Expert-2 program. Once the facts are converted into QUESTIONS, it becomes more difficult to grasp the entire path to be taken to prove or disprove a particular HYPOTHESES. If many facts have been reduced down to DEDUCTIONS, it may be extremely difficult to work "backwards" to see what facts really prove a HYPOTHESES as opposed to what the code (which has a "bug" in it) shows.

4. Future modification of a program may be impossible without good documentation of the original data. Many times in the data processing world, expensive programs are thrown out and re-written because proper guidelines were ignored and the "new" programmer could not make heads or tails out of an improperly documented program.

After the FACTS have been listed, you are ready to proceed to Guideline 3.

GUIDELINE 3: Put these FACTS into chart form.

This is a very straightforward step. In Figure T-2, the previously listed FACTS have been put into a chart format.

	DISEASES						
SYMPTOMS	Meas	Mump	Chkn Pox	Germ Meas	Flu	Whop Coug	Comm Cold
Fever, high, steady			X				
Cough, moderate					X		
No appetite						X	
Fever, none						X	
Cough, severe with whooping						X	
Cough, light at first						X	
Tearing						X	
Sneezing						X	X
Diarrhea					X		
Vomiting					X		
Sore throat					X		X
Sweating					X		
Weakness					X		
Headache, bad					X		
Headache, moderate				X			X
Stiff joints				X			
Enlarged lymph nodes behind ear				X			
Rash, face, neck then body				X			
Rash, rose-colored				X			
Fever, slight				X			
Chills			X		X		
Body aches, mild			X				X
Rash, spotty, pimples and blisters			X				
Enlarged neck lymph nodes		X					
Scant or excessive saliva		X					
Cannot suck lemon		X					
Fever, moderate		X					X
Swelled salivary glands		X					
Rash, on scalp, then body	X						
Rash, brownish pink	X						
White spots inside cheek	X						
Conjunctivitis	X				X		
Bloodshot eyes	X						
Cough, harsh, hacking	X						
Fever, high, fast rising	X				X		
Runny nose	X			X	X	X	X
Stuffy nose, later							X
Body aches, severe					X		

Figure T-2: Chart of Symptoms

All of the FACTS are listed in a column on the left. The HYPOTHESIS are listed in columns to the right. An "X" is placed whenever a FACT is true for a given HYPOTHESES.

The purpose of putting the FACTS into chart format is so that all

of the necessary information is available in one concise location. This makes implementing Guideline 4 much easier.

For larger problems, it may be necessary to make a chart several pages long. For complex problems with many non-related possible HYPOTHESIS, it may be necessary to create several separate charts with each chart reflecting a different portion of the problem.

Once you have written out all of the needed facts and plotted them into a chart format, you are ready for Guideline 4.

GUIDELINE 4: Using the chart you just made, factor or classify FACTS into general categories of "natural groupings."

The purpose of this guideline is to help you to write a much more organized "top-down" style Expert-2 program. The technique used here is very much like the factoring used in high school algebra. What you must do is examine the data in your chart and look for commonality of FACTS between HYPOTHESIS. We will then make another chart composed of these factored, general categories.

In looking at your chart or charts, you can see tens or even hundreds of FACTS. Some of these FACTS should have some commonality among them. By noting this commonality and asking a general QUESTION concerning these FACTS, you can either follow one logic path, asking more detailed QUESTIONS about the same subject or discard a whole subject.

The animal classification program (Figure T-1) contains a splendid example of this type of factoring. A DEDUCTION is drawn whether or not the animal in question is a MAMMAL. Once this is established, whole sections of code can be eliminated or selected by the use of "IF animal is mammal" or "IFNOT animal is mammal."

The childhood diseases program that we will be programming does not have these straight forward kinds of commonality. We chose it because most data bases do not have such clear cut divisions but are still factorable.

This guideline is not a hard and fast rule. If you can factor a lot of your data, that is good. You have simplified your task. If your data is such that it cannot be easily factored, do not worry about it. The programming task can be done either way. But the benefits derived from factoring are great enough that you should attempt at least one "factoring pass" through the data.

Let us now examine our chart (Figure T-2) and look for FACTS which when combined to form DEDUCTIONS will separate whole groups of HYPOTHESIS.

SYMPTOMS	D I S E A S E S						
	Measles	Mumps	Chicken Pox	German Measles	Flu	Whooping Cough	Common Cold
Fever	X	X	X	X	X		X
Rash	X		X	X			
Cough	X				X	X	
Headache				X	X		X
Body ache			X		X		X
Sneezing						X	X
Sore throat					X		X
Chills			X		X		
Conjunctivitis	X				X		
Runny nose	X			X	X	X	X

Figure T-3: General Category Chart of Symptoms

The first grouping that came to mind was "fever." There are six FACTS that have "fever" as their common denominator. We then start a new "General Category Chart" composed of these groups. As you can see in Figure T.3, "fever" is the first entry in our General Category Chart. (The importance of this chart will become more obvious in Guideline 5.)

The second group of FACTS is "rash." This too goes into our General Category Chart. We proceed until as many FACTS as possible are listed. At this stage of the game, we have no idea which groups are or are not going to be of use. So we keep on factoring until we run out of relatively obvious groups. If at a later time we discover we need more groups, we can always go back and look for more obtuse groupings.

One technique you may wish to try is to attempt to physically group FACTS together as you are originally creating the chart. This may make it easier to "see" natural groupings.

The final result is shown in Figure T-3. This General Category Chart combined with our exhaustive chart of FACTS will give us all of the information needed to actually program our problem.

GUIDELINE 5: Develop your logic flow using the "divide and conquer" approach.

There are countless ways to program any moderately sized problem. However, there are easy approaches and more difficult approaches. There are approaches that clearly lead the user to a HYPOTHESES and ones that confuse the user but still arrive at a suitable HYPOTHESES. It is the purpose of this guideline to hopefully aim you in a more logical direction.

In essence, the "divide and conquer" approach is a technique

where you eliminate half of your possible HYPOTHESIS at once by asking a QUESTION which is true for half of them and not true for the other half. You then proceed to "divide and conquer" both of these halves, etc., until the problem is broken down to a level where you can set about proving one individual HYPOTHESES.

Before we start to program our childhood diseases program using the "divide and conquer" approach, let us momentarily pause and examine an alternate "brute force" approach. Doing this will help you to better understand and appreciate the "divide and conquer" method.

The most direct "brute force" method would be to make one RULE for each childhood disease. There would be seven RULES.

For example the RULE for measles would then be:

```
( RULE 1 -- Measles )
IF subject has rash on scalp, then body
AND subject has brownish pink rash
AND subject has white spots inside cheek
AND subject has conjunctivitis
AND subject has bloodshot eyes
AND subject has harsh hacking cough
AND subject has high, fast rising fever
AND subject has runny nose
THENHYP is Measles
```

The next RULE would contain all of the symptoms for mumps, the next for chicken pox, etc. If any QUESTION in a question section is answered negatively, EXPERT-2 will immediately discard that RULE and move on to the next RULE.

At this level the problem is simple and easy to code. It might even be a realistic approach to solving a problem of this size. There are, however, disadvantages to this approach. The user immediately begins answering highly detailed QUESTIONS; lots of QUESTIONS for complex problems. When a negative response to a QUESTION causes Expert-2 to try to prove a new HYPOTHESES, there is no contextual indication to the user that this has happened. All that happens is more, similar, detailed QUESTIONS are asked. The user therefore has difficulty discerning the course you, the Expert, are following.

The "divide and conquer" approach has the advantage of giving the user an inkling of where you are heading through the use of high level questions.

Let us begin solving the childhood disease problem using the "divide and conquer" approach. We will be referring to Figure T-4 which is a graphic illustration of the solution. This figure is called a "decision tree" and is an extremely powerful tool for helping you to visualize the logic paths you are creating.

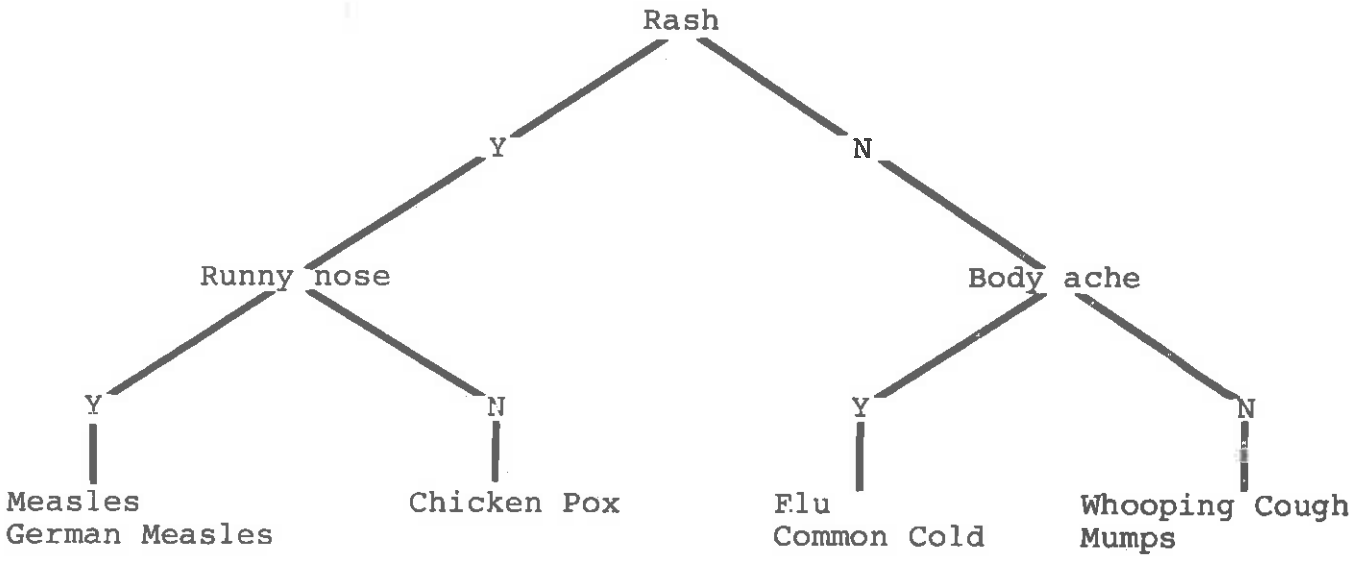


Figure T-4: Decision Tree

We begin coding our childhood diseases Expert-2 program by examining Figure T-3, the General Category Chart of generalized FACTS. Following the "divide and conquer" strategy, we look for a category that will roughly divide the seven possible HYPOTHESIS into two groups.

"Rash" splits the HYPOTHESIS into two groups quite nicely. Three HYPOTHESIS (measles, chicken pox and German measles) have "rash" as a symptom. The remaining four diseases do not. Referring to the decision tree, Figure T-4, we can graphically see "Rash" dividing the problem into two parts.

The next step is to further "divide and conquer" each of the two groups we just created.

Again, referring to the General Category Chart, Figure T-3, we look for a category that will divide those diseases that do have a rash into two groups. Scanning down the list of symptoms, we find that "runny nose" is a symptom that splits those diseases that do have a rash. The "yes there is a rash" portion of the decision tree, Figure T-4, shows us how "runny nose" further divides our possible HYPOTHESIS.

Focusing our attention on the remaining "no rash" HYPOTHESIS/diseases, we see (referring to the General Category Chart, Figure T-3) that "body ache" is a quite reasonable separator. Flu and the Common Cold have associated body aches; Whooping Cough and Mumps do not. As before, the decision tree, Figure T-4, illustrates this further subdividing of the seven HYPOTHESIS.

Another good choice for dividing the "no rash" diseases would have been "sore throat." It might have even been a better choice. This situation is where your expert knowledge would guide you into making the wisest choice. If it is a toss up as to which category to use, choose the one that will give the user the best idea of what you are trying to prove.

OK. Let us see where we are so far. Referring to the decision tree, Figure T-4, we can see that the problem has now been divided into the four separate groups:

1. Measles Types:
   German Measles
   Measles

2. Chicken Pox

3. Flu Types:
   Flu
   Common Cold

4. Non-rash, Non-flu:
   Mumps
   Whooping Cough

Is this enough division? There are still unused categories in our General Category Chart. Should we break the problem down further in order to utilize these remaining categories? Have the possible HYPOTHESIS been categorized to small enough groups? In this case, we have probably broken the problem down enough. But as you can see, this is an objective, not subjective question and you, the expert, will have to make the decision for your particular problem.

Remember though that none of this is "cast into concrete." If, once you start coding RULES, you discover a better way to "divide and conquer"; GO BACK AND REDO IT if it will gain you anything. If you find that you need a finer subdivision of a HYPOTHESES, simply GO BACK AND REDO IT. Although it is more work (sometimes much more work), it will be worth it because you will have a more easily understandable program and a less confused user.

Finally, after meticulous planning of our "strategy" (via our decision tree, Figure T-4) and our "tactics" (via our chart of symptoms, Figure T-2); we are ready for Guideline 6.

GUIDELINE 6: Convert the FACTS into QUESTIONS, i.e., RULES.

At last! We are now going to write the RULES. We have done our homework and laid the foundation. Our "high-level logic" is laid out for us in the form of the decision tree, Figure T-4. We should be able to lead the user through an understandable logic path. Our "low-level logic" is carefully documented in our table of FACTS, Figure T-2. Using this data, we can be assured that we ask the correct and

necessary QUESTIONS to come to the correct HYPOTHESES.

Obviously, we must write code (RULES) that reflects all of the logic paths. As a matter of personal preference, we will follow the left (true) logic path first; then work our way to the right (false).

Using the decision tree, Figure T-4, as a guide; let us write RULE 1.

Before writing a RULE, always ask yourself, "What's the purpose of this RULE?" (This is the same old thing of knowing your goal. Only this time it is at a very detailed level.) So, what do we want RULE 1 to do?

Looking at the decision tree (Figure T-4) we see that if we combined "rash" and "runny nose" into one RULE we could make a DEDUCTION that the disease "might be measles" of one sort or another. Therefore:

```
( RULE 1 -- might be Measles )
IF  subject has rash
AND  subject has runny nose
BECAUSE  rash and runny nose might mean measles
THEN  might be measles
```

Note the BECAUSE statement explains to the user why you are asking these particular QUESTIONS.

Referring to Figure T-4 you can see that this particular logic path has answered enough questions to led us to one of three HYPOTHESIS; namely measles, German measles or chicken pox. At this point, let us create an intermediate level RULE that will use the already answered QUESTIONS to ascertain if the disease in question "might be chicken pox." This intermediate level RULE is not absolutely necessary for this problem but it shows the usefulness of factoring your problem with RULES. This leads us to RULE 2:

```
( RULE 2 -- might be Chicken Pox )
IF  subject has rash
ANDNOT  subject has runny nose
BECAUSE  no runny nose with chicken pox
THEN  might be chicken pox
```

Notice that this RULE uses an ANDNOT. The DEDUCTION "might be chicken pox" can only be true if the user previously answered "N" to the question "subject has runny nose."

Also notice that the DEDUCTION drawn is that the disease "might be chicken pox." This is a very important point! According to the decision tree (Figure T-4), once we are "at this level" if the disease is not a form of measles, it must be chicken pox. Right? Wrong! A rash and no runny nose does not necessarily mean chicken pox. What about poison oak? Eventually we must substantiate our conclusion of "might be chicken pox" with enough specific QUESTIONS that we can make a HYPOTHESES with certainty that the disease is undeniably chicken

pox. Be careful when programming that you do not fall into this type of trap. You must always use enough FACTS to conclusively prove a given HYPOTHESES to be true.

On to RULE 3. This particular problem (childhood diseases) is simple enough that we can now attempt to definitely prove that either measles or German measles is the disease in question. More complex problems, e.g., repairing an aircraft jet engine, may require many more levels of intermediate RULES. But here, we can now ask enough detailed QUESTIONS to "prove" measles to be the disease.

```
( RULE 3 -- is Measles )
IF  might be measles
AND  subject has brownish pink colored rash
BECAUSE  measles have brownish pink rash
AND  subject has rash on scalp, then body
AND  subject has white spots inside cheek
AND  subject has conjunctivitis
AND  subject has bloodshot eyes
AND  subject has harsh, hacking cough
AND  subject has fast rising, high fever
THENHYP  is Measles
```

If Expert-2 cannot prove RULE 3 to be true, i.e., the subject does not have measles; then Expert-2 will attempt to prove the next HYPOTHESES. According to our decision tree (Figure T-4), this next HYPOTHESES should attempt to prove, one way or another, whether the subject has German measles.

```
( RULE 4 -- is German Measles )
IF  might be measles
AND  subject has rose colored rash
BECAUSE  German measles rash is rose colored
AND  subject has rash on face, then body
AND  subject has moderate headache
AND  subject has stiff joints
AND  subject has enlarged lymph nodes behind ear
AND  subject has slight fever
THENHYP  is German Measles
```

The next step is to trace the false path (in our decision tree, Figure T-4) and follow up on RULE 2 -- "might be chicken pox."

```
( RULE 5 -- is Chicken Pox )
IF  might be chicken pox
AND  subject has high, steady fever
AND  subject has chills
AND  subject has mild body aches
AND  subject has rash, spotty, pimples and blisters
THENHYP  is Chicken Pox
```

We have now written three HYPOTHESIS RULES for: German measles, measles and chicken pox. (Remember that HYPOTHESIS RULES must contain a THENHYP.) Now we must take the false logic path for "subject has rash." Just as we combined "subject has rash" and "subject has runny

nose" in RULE 1, so can we combine IFNOT "subject has rash" and "subject has body ache" to separate out the flu-type diseases.

```
( RULE 6 -- might be flu/cold )
IFNOT  subject has rash
AND  subject has body aches
BECAUSE  body aches tend towards flu or cold
THEN  might be flu/cold
```

We can now use the DEDUCTION "might be flu/cold" to definitely prove if the subject has the flu or a common cold.

```
( RULE 7 -- is Flu )
IF  might be flu/cold
AND  subject has moderate cough
AND  subject has possible diarrhea
AND  subject has possible vomiting
AND  subject has sore throat
AND  subject has sweating
AND  subject has weakness
AND  subject has bad headache
THENHYP  is FLU

( RULE 8 -- is Common Cold )
IF  might be flu/cold
AND  subject has sneezing
AND  subject has sore throat
AND  subject has moderate headache
AND  subject has mild body aches
AND  subject has moderate fever
AND  subject has runny nose
THENHYP  is Common Cold
```

The last two RULES that we need to code are for either positive identification of Mumps or Whooping Cough.

```
( RULE 9 -- is Mumps )
IFNOT  subject has rash
IFNOT  might be flu/cold
AND  subject has swelled salivary glands
AND  subject has enlarged neck lymph nodes
AND  subject has scant or excessive saliva
AND  subject cannot suck lemon
AND  subject has moderate fever
THENHYP  is Mumps

( RULE 10 -- is Whooping Cough )
IFNOT  subject has rash
IFNOT  might be flu/cold
AND  subject has no appetite
AND  subject has no fever
AND  subject has cough, light at first
AND  subject has severe whooping cough
AND  subject has tearing
AND  subject has sneezing
```

```
AND  subject has runny nose
THENHYP  is Whooping Cough
```

Notice the IFNOT "subject has rash." In originally testing these RULES, we found that this statement was necessary. Without it, a negative answer to the final test for measles (RULE 5) would cause Expert-2 to try proving this RULE instead of responding with "CANNOT PROVE ANYTHING."

That completes our Expert-2 program for determining childhood diseases. The following is our completed code in its entirety:

```
( RULE 1 -- might be Measles )
IF  subject has rash
AND  subject has runny nose
BECAUSE  rash and runny nose might mean measles
THEN  might be measles

( RULE 2 -- might be Chicken Pox )
IF  subject has rash
ANDNOT  subject has runny nose
BECAUSE  no runny nose with chicken pox
THEN  might be chicken pox

( RULE 3 -- is Measles )
IF  might be measles
AND  subject has brownish pink colored rash
BECAUSE  measles have brownish pink rash
AND  subject has rash on scalp, then body
AND  subject has white spots inside cheek
AND  subject has conjunctivitis
AND  subject has bloodshot eyes
AND  subject has harsh, hacking cough
AND  subject has fast rising, high fever
THENHYP  is Measles

( RULE 4 -- is German Measles )
IF  might be measles
AND  subject has rose colored rash
BECAUSE  German measles rash is rose colored
AND  subject has rash on face, then body
AND  subject has moderate headache
AND  subject has stiff joints
AND  subject has enlarged lymph nodes behind ear
AND  subject has slight fever
THENHYP  is German Measles

( RULE 5 -- is Chicken Pox )
IF  might be chicken pox
AND  subject has high, steady fever
AND  subject has chills
AND  subject has mild body aches
AND  subject has rash, spotty, pimples and blisters
THENHYP  is Chicken Pox
```

```
( RULE 6 -- might be flu/cold )
IFNOT  subject has rash
AND  subject has body aches
BECAUSE  body aches tend towards flu or cold
THEN  might be flu/cold

( RULE 7 -- is Flu )
IF  might be flu/cold
AND  subject has moderate cough
AND  subject has possible diarrhea
AND  subject has possible vomiting
AND  subject has sore throat
AND  subject has sweating
AND  subject has weakness
AND  subject has bad headache
THENHYP  is Flu

( RULE 8 -- is Common Cold )
IF  might be flu/cold
AND  subject has sneezing
AND  subject has sore throat
AND  subject has moderate headache
AND  subject has mild body aches
AND  subject has moderate fever
AND  subject has runny nose
THENHYP  is Common Cold

( RULE 9 -- is Mumps )
IFNOT  subject has rash
IFNOT  might be flu/cold
AND  subject has swelled salivary glands
AND  subject has enlarged neck lymph nodes
AND  subject has scant or excessive saliva
AND  subject cannot suck lemon
AND  subject has moderate fever
THENHYP  is Mumps

( RULE 10 -- is Whooping Cough )
IFNOT  subject has rash
IFNOT  might be flu/cold
AND  subject has no appetite
AND  subject has no fever
AND  subject has cough, light at first
AND  subject has severe whooping cough
AND  subject has tearing
AND  subject has sneezing
AND  subject has runny nose
THENHYP  is Whooping Cough
```

Introduction to Programming in FORTH

If you wish to expand the power of Expert-2 through the use of "analytical subroutines," you must program those subroutines using FORTH. Fortunately, it is relatively easy for a novice to use FORTH to perform simple tasks.

Let us briefly review "analytical subroutines." An "analytical subroutine" is a user coded extension of Expert-2. You "call" a subroutine via the "RUN" suffix appended to a command; IFRUN, ANDRUN, and THENRUN. The subroutine name must follow the command. The purpose of the subroutines is to allow you to customize a general purpose Expert-2 system to exactly fit your application.

The use of subroutines falls into two broad categories: 1) asking more detailed QUESTIONS (via IFRUN and ANDRUN) than is possible just with IF and AND and 2) outputting additional information to the user (via THENRUN). Of course, subroutines can be made to perform any other tasks depending upon how much effort goes into their programming.

SUPER-FORTH is a language composed of "definitions" which are commonly referred to as "words." Each "word" has a name. Typing that name on the console or including the name within a new definition is all that is needed to "execute" that definition.

Each word is made up of other "lower level" words that already exist. These words can be either SUPER-FORTH "nucleus" words supplied with SUPER-FORTH, words that you have previously created yourself or a combination of both. (The very lowest level FORTH words, called primitives, are actually made up of the machine language code for the processor they are running on. This fact will be transparent to you.)

It is quite easy to create a new word to perform some specific purpose. To tell SUPER-FORTH that you are defining a new word, you start out with a : (a colon) preceded and followed by at least one space. (FORTH uses spaces as delimiters.) After the : comes the name of the new word. Following the name come previously defined words that will do whatever you want done. You end a definition with a ; (a semicolon). This tells FORTH that the definition is finished.

Let us look at an example.

```
: DO-ALL   DO-1ST  DO-2ND  DO-3RD  ;
```

In this example, the : tells FORTH that we want to create a new word and the name of this new word is to be "DO-ALL." When DO-ALL executes, three previously defined words; DO-1ST, DO-2ND and DO-3RD are to sequentially execute. Finally the ; tells FORTH to stop defining the new word.

Now simply using the new word DO-ALL will do DO-1ST, then DO-2ND and finally DO-3RD. If desired, you could now use DO-ALL in an even "higher level" word. For example:

```
: DO-EVERYTHING
  START-UP
  DO-ALL
  DO-REST
  FINISH-UP ;
```

Outputting Additional Information

Let us make a FORTH word that can be called as a subroutine to output additional information to the user.

```
: MOREINFO  CR  ." Any text following the ´dot quote´
  word up to, but not including a terminating quote
  mark, will be output to the CRT. " ;
```

The word starts with a : and is named MOREINFO. The SUPER-FORTH word CR (pronounced "carriage return" or "c-r") skips to a new line. (It outputs a "carriage return" command, hence the name CR.) The word ." (pronounced "dot-quote") is the SUPER-FORTH word that prints all of the text following it until the terminating quote mark. ." will output a character string up to 127 characters. (If you need more characters than 127, simply close the first ." with its terminating " and add another CR and start a new ." statement.) Note that the single tick marks (´) are not counted as terminating quote marks. Finally, a ; is used to tell SUPER-FORTH that the word is finished.

When MOREINFO is called via a "THENRUN MOREINFO" statement, the text will be output to the CRT.

THE STACK

Many operations of SUPER-FORTH centers around what is called the parameter "stack." If you are familiar with the operation of Hewlett-Packard calculators, you already know how to use FORTH´s stack. If not, the stack is easy to understand.

The stack is used to hold parameters--numbers or data. Words normally pass parameters to each other via the stack. Picture the stack as one of those plate holders you see in a cafeteria. (See Figure T-5.) The last plate put into the holder is the first plate removed. Likewise the last value put onto the stack is the first value removed. You do not have to directly concern yourself with putting parameters onto or taking them off of the parameter stack. SUPER-FORTH does that automatically for you.

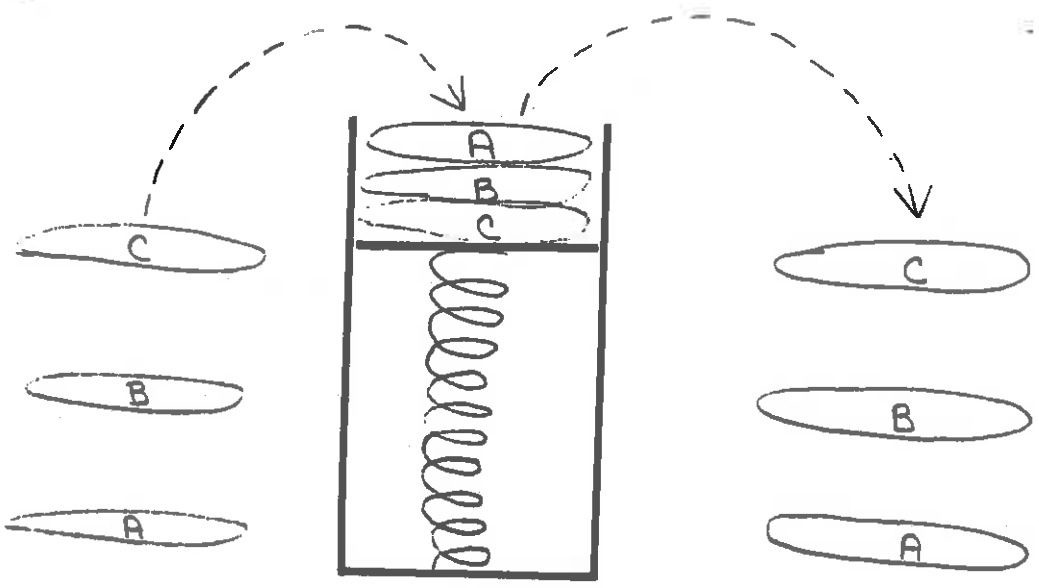


Figure T-5: Parameter Stack

Executing a word that requires a parameter automatically "gobbles up" the top stack parameter. If two parameters are required; two parameters are "gobbled." Likewise, if a word returns a parameter, that parameter is automatically put onto the parameter stack.

There is an SUPER-FORTH word called .S that will "non-destructively" list the contents of the parameter stack onto the CRT. This is a very useful word to use when you are debugging your subroutines. Let us try .S and see how it works.

First let us make sure the stack is empty. The easiest way to do this is to type a few characters of gibberish onto the CRT and then press carriage return. SUPER-FORTH will attempt to locate this gibberish in its dictionary and will fail. When it fails, it will display the error message "NOT RECOGNIZED" underneath the characters in question. Secondly SUPER-FORTH will empty all parameters from its parameter stack. Try it.

Enter:

```
LKJL (cr)
```

(providing, of course, you have not previously defined a word named LKJL!)

Now enter:

```
.S (cr)
```

SUPER-FORTH will respond with "EMPTY STACK."

Now put some numbers onto the stack by entering:

1 2 3 4 (cr) (Each number separated by a space)

Now type:

.S (cr)

SUPER-FORTH will respond with 4 3 2 1 where the leftmost number is the value on the "top" of the stack.

To print out a value on the top of the stack use the SUPER-FORTH word . (a period called "dot"). Type:

. (cr)

SUPER-FORTH responds with 4.

Remember that the last value entered is the first value removed.

Try playing with the stack a while until you become familiar and comfortable with its operation.

Stack Operators

Let us now go over the operation of a few simple stack operators.

+ (valuel \ value2 --- sum)
+ (pronounced "plus") adds the top two values on the stack and replaces them with their sum.

- (minuend \ subtrahend --- difference)
- (pronounced "minus") subtracts the top stack entry from the second stack entry and replaces both values with their difference.

* (valuel \ value2 --- product)
* (pronounced "times") multiplies the top two stack entries together and replaces both values with their product.

/ (dividend \ divisor --- quotient)
/ (pronounced "divide") divides the second stack entry by the top stack entry and replaces both values with their dividend.

/MOD (dividend \ divisor --- remainder \ quotient)
/MOD (pronounced "divide-mod") divides the second stack entry by the top stack entry and replaces both values with their dividend and their remainder. The remainder takes its sign from the dividend.

= (value1 \ value2 --- boolean truth flag)
= (pronounced "equals") compares the top two stack entries and replaces them with a "truth flag" of "1" if the two are equal and a "0" value if the two are not equal.

> (value1 \ value2 --- boolean truth flag)
> (pronounced "greater-than") compares the top two stack entries and replaces them with a "truth flag" of "1" if the second stack entry is less than the top stack entry and a "0" if the second stack entry is equal to or greater than the top entry.

< (value1 \ value2 --- boolean truth flag)
< (pronounced "less-than") compares the top two stack entries and replaces them with a "truth flag" of "1" if the second stack entry is greater than the top stack entry and a "0" if the second stack entry is equal to or less than the top entry.

0= (value --- boolean truth flag)
0= (pronounced "zero-equals") examines the value on the top of the stack and replaces it with a true flag "1" if the value is equal to 0 or with a false flag "0" if the value is not equal to 0.

0< (value --- boolean truth flag)
0< (pronounced "zero-less-than") examines the value on the top of the stack and replaces it with a true flag "1" if the number is less than zero (negative) or with a false flag "0" if the number is greater than or equal to zero.

0> (value --- boolean truth flag)
0> (pronounced "zero-greater-than") examines the value on the top of the stack and replaces it with a true flag "1" if the number is greater than zero (positive) or with a false flag "0" if the number is less than or equal to zero.

Notice how all of the above SUPER-FORTH words always "gobble up" all of their input parameters. This is an almost universally followed rule of thumb for FORTH: A word "uses up" all of its inputs from the parameter stack.

There are many times when this "gobbled up" data is <u>also</u> needed as input for other words. The following stack operators are provided to support just such instances.

DROP (value to be dropped ---)
DROP (pronounced "drop") discards the value on the top of the stack.

DUP (value1 --- value1 \ value1)
DUP (pronounced "dupe") duplicates the top value on the stack. This is useful if you want to keep a certain value and also feed it to a word. Just DUP the value first.

OVER (value2 \ value1 --- value2 \ value1 \ value2)
OVER (pronounced "over") copies the second stack entry onto the top of the stack.

ROT (value1 \ value2 \ value3 -- value2 \ value3 \ value1)
ROT (pronounced "rote") rotates the stack by removing the third stack entry and placing it onto the top of the stack.

SWAP (value2 \ value1 --- value1 \ value2)
SWAP (pronounced "swap") exchanges the top two values on the stack.

Go ahead and practice using these words while observing the results via .S.

Constants and Variables

Constants and variables in SUPER-FORTH are called just that: CONSTANT and VARIABLE. They are used in the form:

value CONSTANT name

VARIABLE name (Note no initial value)

Executing the name of a CONSTANT places the value of the CONSTANT onto the top of the parameter stack.

Executing the name of a VARIABLE places the address of the variable's value onto the top of the parameter stack.

Memory Access

Memory access is performed via the FORTH words @ and ! (pronounced "fetch" and "store" respectively).

@ (memory address --- data)
@ (pronounced "fetch") gobbles an address and replaces it with the 16-bit memory contents of that address.

! (data \ memory address ---)
! (pronounced "store") does the opposite of @. ! gobbles both the data and an address from the stack and "stores" the 16-bit memory location (pointed to by the address) with the supplied data.

The following are examples of memory accesses:

To create a VARIABLE named COUNT, you would type:

```
VARIABLE COUNT (cr)
```

To fetch data from the VARIABLE named COUNT, you would type:

```
COUNT @ (cr)
```

To store the data 1234 into COUNT, you would type:

```
1234 COUNT ! (cr)
```

To store a CONSTANT named COUNT-VAL equaling the value 1000, you would first have to create the CONSTANT by typing:

```
1000 CONSTANT COUNT-VAL (cr)
```

Then store the CONSTANT's value into the VARIABLE COUNT by typing:

```
COUNT-VAL COUNT ! (cr)
```

Logical Operators

These words perform "logical functions" on stack values. What are "logical functions" you ask? The two that you would use most often are:

OR (value1 \ value2 --- boolean truth flag)
OR (pronounced "or") replaces the top two stack values with their boolean sum. That is, if either one or both of the two values are TRUE (non-zero), the resulting truth flag will be TRUE (1). If both values are FALSE (zero), then the truth flag will be FALSE (0). This action is identical to the English language concept of "or."

AND (value1 \ value2 --- boolean truth flag)
AND (pronounced "and") replaces the top two stack values with their boolean product. That is, if both values are TRUE (non-zero), the resulting truth flag will be TRUE (1). If any one or both values are FALSE (zero), the resulting truth flag will also be FALSE (zero). This action is identical to the English language concept of "and."

Control Words

SUPER-FORTH has an extensive array of control-type words that allow conditional decisions to be made (e.g., IF, ELSE, THEN) and different types of looping structures. The use of these words is beyond the scope of this brief introduction to SUPER-FORTH programming.

The books mentioned in the bibliography, especially STARTING FORTH and ALL ABOUT FORTH are excellent teaching tools for learning FORTH programming. Both of these books are available through Mountain View Press.

Putting It All Together

Let us use some of these SUPER-FORTH words to actually write a subroutine to input additional data.

The purpose of this subroutine will be to avoid the ambiguity of what a "mild," "moderate," or "severe" fever actually is. For purposes of this example, we shall ask the user to type the temperature in question into the computer. (in a real-life problem you might just ask if the fever was "between 98 and 100 degrees" or "between 100 and 102 degrees," etc.

What we need to do is to write five simple subroutines. One will ask the user to type in the temperature and will then save it in a variable. The remaining four subroutines will determine if the fever is mild, moderate, severe or if there is no fever at all. The fact that we must have four separate subroutines is dictated to us by Expert-2. Remember that we are replacing the QUESTION asked by an IF or ANDIF statement that returns a single truth flag whenever it executes. Since we have four questions and only one truth flag returned each time, we obviously must have four subroutines.

To begin we must first define the Expert-2 code that will call the subroutines. What is it we want to do? We want to come to one of four DEDUCTIONS via THEN statements. The CONCLUSION being either no, mild, moderate or severe temperature.

Since we can give the subroutines any name we want, let us give them understandable, readable names that define their functions at a glance:

```
GET-TEMP
?NO-TEMP
?MILD-TEMP
?MODERATE-TEMP
?SEVERE-TEMP
```

(Standard FORTH naming convention says that the name of a word which does nothing but return a truth flag should begin with a question mark.)

Now that we have named the subroutines, let us write the high level Expert-2 code to call them.

We only have to ask the user to enter the temperature once. Therefore, we need only to call GET-TEMP from one RULE, the first RULE.

```
( Rule 1 )
IFRUN   GET-TEMP
ANDRUN  ?NO-TEMP
THEN    no fever

( Rule 2 )
IFRUN   ?MILD-TEMP
THEN    fever is mild

( Rule 3 )
IFRUN   ?MODERATE-TEMP
THEN    fever is moderate

( Rule 4 )
IFRUN   ?SEVERE-TEMP
THEN    fever is severe
```

When RULES 1, 2, 3 and 4 execute, one of the DEDUCTIONS will be proven true and three will be proven false. We could have embedded GET-TEMP within the first subroutine called ?NO-TEMP but we have more control of the program for future modifications if we call GET-TEMP directly from the Expert-2 code.

Let us turn our attention to the subroutines themselves and define exactly what we expect each one to do.

GET-TEMP	Type a message to the user asking him to enter the temperature in whole degrees with no decimal points. Save this temperature in the variable SAVED-TEMP. Return a TRUE flag.
?NO-TEMP	Compare the value in SAVED-TEMP with 98 degrees and return a TRUE flag if it equals; a FALSE flag if not.
?MILD-TEMP	Test to determine if the value in SAVED-TEMP is 99 or 100 degrees. Return a TRUE flag if yes, a FALSE flag if not.
?MODERATE-TEMP	Test to determine if the value in SAVED-TEMP is 101 or 102 degrees. Return a TRUE flag if yes; a FALSE flag if not.
?SEVERE-TEMP	Test to determine if the value in SAVED-TEMP is 103 degrees or higher. Return a TRUE flag if yes, a FALSE flag if not.

GET-TEMP

We start our subroutines with GET-TEMP. GET-TEMP gets the temperature from the user via the word INPUT. The word INPUT is a very common, useful definition used to input a single-precision number from the console. It is very similar in function to the INPUT command bound in BASIC. The stack activity in parenthesis shows that the word takes nothing from the stack but returns a single-precision value.

```
( INPUT  GET-TEMP                                    84AUG16 )

1 CONSTANT TRUE
0 CONSTANT FALSE

VARIABLE SAVED-TEMP

:  INPUT  ( --- N )
    QUERY BL WORD NUMBER DROP ;

: GET-TEMP  ( --- FLAG )
   CR  ." Please enter temperature in whole degrees"
   CR  ." without a decimal point:  "
   CR  INPUT  SAVED-TEMP !
   TRUE  ;
```

Note the definitions of TRUE and FALSE. Use of these CONSTANTS makes the subroutines more readable.

INPUT is beyond the scope of this tutorial.

As you can see, GET-TEMP is very straightforward. The CR skips to a new line. The message is output via ." ("dot-quote"). INPUT ends up with the value on the top of the stack. SAVED-TEMP sets up for the ! ("store") by putting the address of the variable on the stack. The ! uses the address to store the value into the variable. Lastly, TRUE puts a true flag onto the stack to satisfy the requirement that all Expert-2 IF statements must return a flag.

?NO-TEMP

?NO-TEMP is shown as follows:

```
: ?NO-TEMP  ( --- FLAG )
   SAVED-TEMP @ 98 = ;
```

Again, the word takes nothing from the stack and returns a flag. SAVED-TEMP sets up for the @ ("fetch") by putting the variable address onto the top of the stack. The @ gobbled the address and copies the previously saved value onto the stack. (The value still remains unchanged in SAVED-TEMP.)

The value 98 is then put onto the stack so a comparison can be made via the following =. The = gobbles up both the fetched value and

the 98 and returns a truth flag. This truth flag will be TRUE if the values were equal and FALSE if they were not. No additional flag is needed since a flag is returned (and put onto the top of the stack) by =.

?MILD-TEMP

?MILD-TEMP is defined as follows:

```
: ?MILD-TEMP  ( --- FLAG )
   SAVED-VALUE @ 99 =
   SAVED-VALUE @ 100 =
   OR ;
```

?MILD-TEMP uses a simple, "brute-force" approach to test for 99 or 100 degrees by first testing for 99 then for 100 and then combining the two resulting truth flags with an OR. The fetching and comparisons are identical to what was done in ?NO-TEMP.

Note that it would be possible to do one fetch from SAVED-TEMP and then use DUP and SWAP to get the value and flags into their proper positions. While possibly executing a little faster, this style of coding results in far less readable code. Hence the approach taken in the example.

The OR at the end of ?MILD-TEMP performs an OR function of the two truth flags left by the =´s. Since an OR will result in a TRUE flag if either or both of its inputs are TRUE, a TRUE condition for either 99 or 100 degrees results in a TRUE output from ?MILD-TEMP.

?MODERATE-TEMP

?MODERATE-TEMP is defined as follows:

```
: ?MODERATE-TEMP  ( --- FLAG )
   SAVED-VALUE @ 101 =
   SAVED-VALUE @ 102 =
   OR ;
```

?MODERATE-TEMP is identical to ?MILD-TEMP except that comparisons are made on 101 and 102 degrees.

?SEVERE-TEMP

?SEVERE-TEMP and NOT are defined as follows:

```
: NOT  ( FLAG --- OPPOSITE FLAG )
   0= ;

: ?SEVERE-TEMP ( --- FLAG )
   SAVED-TEMP @ 103 < NOT ;
```

?SEVERE-TEMP works on the premise that if the value in SAVED-TEMP is "NOT less than 103," it must be equal to or greater than 103. Once you understand this point, the rest is easy. The value is again fetched from SAVED-TEMP by @. Putting 103 on the stack sets up for < ("less-than").

Now comes a slightly tricky part. The < returns a TRUE flag if the value is less than 103 (which is exactly what it should do). Only we want a TRUE flag if 103 is NOT less than the value. How do we change the flag? With the NOT word! NOT simply says if the flag on the top of the stack is false (i.e., 0), return a TRUE flag, else return a FALSE flag (i.e., the truth flag was non-zero).

Using the FORTH EDITOR

The FORTH EDITOR is used to edit source data for MVP-FORTH and Expert-2 programs. It is a "screen-oriented" editor as opposed to the "file-oriented" editor you may be used to using. This is because of the way FORTH handles its data stored on disk.

The more common computer languages keep data stored on disk in the form of "files." Where a file is usually considered to be a sequential collection of data. Files are accessed by names. For example, to edit the source code for a program named "CHILDHOOD," you might type: "EDIT CHILDHOOD."

The FORTH editor, on the other hand, does not use files. Instead it uses "screens" or "blocks." FORTH divides its mass storage up into 1024 byte "blocks." Hence a 256K disk will contain 256 blocks. MVP-FORTH definitions and/or Expert-2 RULES are edited onto these blocks or "screens." The word block is generally used when referring to the physical 1024-byte entities, "blocks," while "screens" generally refer to blocks which contain source code and are edited by the editor.

LOAD

Once you have edited your source code onto screens, you use the MVP-FORTH words LOAD or THRU to "load" the code into the computer. LOAD is used in the form:

```
scr# LOAD (cr)
```

where scr# is a screen (block) number. This will cause one screen (block) to be LOADed and compiled.

For example to load screen 100 you would type:

```
100 LOAD (cr)
```

THRU

THRU is an extension of LOAD. THRU causes multiple sequential screens to be LOADed. THRU is used in the form:

```
beginning-scr# ending-scr# THRU (cr)
```

where the beginning and ending screen numbers specify (inclusively) the range of screens to be sequentially LOADed and compiled.

To load screens 100 through 105, you would type:

```
100 105 THRU (cr)
```

Creating a LOAD Screen

It is highly desirous to create what is called a "load screen." A "load screen" is simply a screen which itself contains LOAD and THRU commands instead of source code. When a load screen is LOADed via a LOAD command, (or multiple load screens via a THRU command), the LOADs and THRUs contained on these screens are executed and the appropriate source code is compiled. This approach saves you from repeatedly having to enter large amounts of data just to get your program compiled. Figure T-6 is an example of a load screen.

```
( CHILDHOOD DISEASE LOAD SCREEN                MVP 840829 )

: WALL ;
20  30 THRU   ( SUBROUTINES )
    54 LOAD   ( INPUT )

RULES
70  80 THRU   ( RULES 0-10 )
    81 LOAD   ( RULE 11 )
    83 LOAD   ( RULE 12 )
90 103 THRU   ( RULES 13-17 )

DONE
```

Figure T-6: Typical LOAD Screen

The Comment Line

A screen is composed of 16 lines of 64 characters each. The first line of a screen is normally reserved as the "comment line." The comment line should begin with a ((left parenthesis) for "start comment" and ends with a) (right parenthesis) for "stop comment." The text on this line usually reflects what source code is on the screen. Often the date and the person editing initials are placed in the righthand portion of the comment line. (is used in the form:

```
( THIS LINE WILL BE A COMMENT LINE. ) (cr)
```

INDEX

The significance of the comment line becomes apparent when you use the word INDEX. INDEX is used in the form:

```
beginning-scr#  ending-scr#  INDEX (cr)
```

where the beginning and ending screen numbers specify (inclusively) a sequential range of screens whose comment lines (0 or topmost line) are to be listed on the CRT. If your comment lines reflect what is on the screens, INDEX will give you a good picture of what and where your

source data is.

If you wish to see what is on the top line of screens 100 through 120, just type:

100 120 INDEX (cr)

The Editor

The MVP-FORTH editor is a very simple "line" editor. It allows you to edit one line at at time. In terms of editors, this editor is very dumb. There are much more sophisticated, less error-prone editors available. This editor was chosen for MVP-FORTH because:

1. It uses the same editor commands as listed in Starting FORTH.
2. It is transportable between different computer systems without any customization being necessary.
3. It does not have to be used with a cursor-addressable console.
4. It is relatively small and compact.

Editor Commands

EDITOR

The editor lives in its own "vocabulary," therefore you must enter this vocabulary's name before MVP-FORTH can recognize any editor commands. This is very easy to do. Simply type the word EDITOR. If you type an editor command and the system responds with a "NOT RECOGNIZED" message, type EDITOR and try again. Note: Whenever you compile any FORTH or Expert-2 definitions, e.g., via a LOAD or THRU, the system exits the EDITOR vocabulary. To start editing again, simply type EDITOR. EDITOR is used in the form:

EDITOR (cr)

Since the editor is written in MVP-FORTH, editor commands are also MVP-FORTH words and therefore must follow the same syntax rules. The point to keep in mind is that individual commands must be separated by at least one space and commands do not execute until a carriage return is entered to terminate the command line.

Screen-Oriented Commands

LIST

To begin any "editing session," you must first list the screen to

be edited via the word LIST. LIST is used in the form:

scr# LIST (cr)

For example, to LIST screen 100 you would type:

100 LIST (cr)

If you type LIST without first putting the desired screen number on the stack, whatever value that is on the stack will be used. There is no telling what will be displayed. If the number is greater than the number of blocks MVP-FORTH "thinks" are on the current drive, you will get the error message "BLOCK NUMBER OUT OF RANGE." If this happens, just try again making certain to put the number of the screen on the stack before typing LIST.

L Command

Once you list a screen via the word LIST, that screen number is saved by the editor and you can relist that screen simply by typing the command L. L also shows what line you are currently editing if any and your current cursor location. The L command is used in the form:

L (cr)

At this point in an editing session, typing L would relist screen 100.

N Command

Once a screen is listed, it is possible to list the next sequential screen by typing the editor command N (for "next") followed by a space and the command L to actually list the screen. The N command is usually used in the form:

N L (cr)

At this point in an editing session, typing N L would list screen 101.

B Command

Conversely, to list the previous sequential screen, type B (for "before") followed by space and L to list the screen. The B command is usually used in the form:

B L (cr)

At this point in an editing session, typing B L would list screen 100 again.

FLUSH

When you are finished editing a particular screen, you must write that changed screen back onto the disk. This is done via the FLUSH command. FLUSH will write to disk any blocks/screens which have been modified or changed. If you do not FLUSH a modified screen, it will probably only remain in memory. If you turn off the computer or reboot, you will lose that data. FLUSH is used in the form:

FLUSH (cr)

If you had made editing changes to screen 100, FLUSH would update the disk with that new screen.

EMPTY-BUFFERS

If you happen to confuse a screen so badly that you simply want to quit and start over, you can erase all of the screens in memory with EMPTY-BUFFERS. Once you have emptied the buffers, you must relist any screen you desire to edit. Of course, to be effective, EMPTY-BUFFERS must be done in place of FLUSH! EMPTY-BUFFERS is used in the form:

EMPTY-BUFFERS (cr)

If you had made editing changes to screen 100, EMPTY-BUFFERS would cause you to have a fresh copy of screen 100 before the last set of changes were made.

Before changing disks, rebooting or powering off the computer, always do one of the following:

1. Type FLUSH -- to save any existing data.

2. Type EMPTY-BUFFERS -- to empty the current buffers.

COPY

Sometimes it is necessary to copy the contents of one screen to another screen. This can be accomplished through the use of COPY. COPY is used in the form:

from-scr# to-scr# COPY (cr)

Note that the from (source) screen is second on the stack and the to (destination) screen is on the top. If you switch these, you will clobber the very screen you want to duplicate. Be careful! (Also do not forget to FLUSH afterward!)

To copy screen 100 onto screen 101, you would type:

```
100 101 COPY FLUSH (cr)
```

WIPE

There is one last standard screen-oriented command. It is often necessary to clear a screen to blanks. Although it is possible to clear a screen line by line, it is much easier to use the WIPE command. This command is used in the form:-

```
WIPE (cr)
```

The screen that was LISTed last (either through LIST or L) will be "wiped" to blanks. (Do not forget to do a FLUSH.)

To be safe, always type the L command (and carriage return) before typing in the word WIPE. That way you will be certain which screen is going to be erased.

TIDY

There is one non-standard screen-oriented command that is sometimes very useful. This command is called TIDY. Since this command is not included with SUPER-FORTH, it is defined in Figure T-6. Its purpose is to "tidy" up a screen that may contain "control characters." "Control characters" are undisplayable characters that, due to the simplicity of the SUPER-FORTH editor, are sometimes put into a screen. You cannot see these characters when you LIST a screen, but when you try to LOAD that screen, the compiler does see these characters and stops.

If you seem to be having problems with a particular screen in that the source code does not load properly, try running TIDY in the form:

```
scr# TIDY (cr)
```

Do not forget to do a FLUSH.

```
( ?NOT-PRINTABLE  TIDY )

: ?NOT-PRINTABLE ( CHAR --- FLAG )
   DUP BL <  SWAP 126 >  OR  ;

: TIDY ( SCREEN# --- )
   BLOCK 1024 OVER + SWAP
   DO    I C@ ?NOT-PRINTABLE
         IF   BL I C!  UPDATE
         THEN
   LOOP ;
```

Figure T-6: Source Code for TIDY

SUGGESTED READING LIST

All About FORTH by Glen B. Haydon, Mountain View Press.

MVP-FORTH Source Listings by G. Haydon and R. Kuntze, Mountain View Press.

Starting FORTH by Leo Brodie, Prentice-Hall.

Thinking FORTH by Leo Brodie, Prentice-Hall.

All of the above books and other FORTH books may be obtained from:

MOUNTAIN VIEW PRESS
P. O. Box 4656
Mountain View, CA 94040

INDEX

Now, you can begin building you own EXPERT programs. Let's see how we can help each other. We

would like to do several things for users of EXPERT Systems.....

Referral service for contract jobs

Cross pollination of ideas and programs

Comments and corrections

Keep you aware of future developments

We can only do these things if we know something about you. Please answer the following questions and return the form to:

PARSEC RESEARCH
Drawer 1766
Fremont, CA 94538

1) Name_____

Address_____

City_____State____ZIP_____

Phone (_____) _____

2) Primary interest in EXPERT systems_____

_____

_____

3) Specific program being developed_____

_____

4) Interested in: ___Referrals ___Exchanging Ideas
___Marketing outlet for programs ___Other _____

_____