The Internals of FORPS: A FORth-based Production System

Christopher J. Matheus

University of Illinois
Urbana, Illinois

Abstract

Two distinct programming methodologies—production systems and FORTH—have been combined to form a unique system called FORPS. Production systems provide a practical and efficient means for structuring knowledge applicable to the intelligent solution of specific problems. These systems have been widely exploited by the Artificial Intelligence community in the creation of expert systems to assist in such tasks as the diagnosis of disease, the design of computer systems, and the search for natural resources. FORTH has found its domain largely in its application to real-time control problems. Many of these applications are becoming increasingly difficult and complex, and require at least a moderate degree of intelligence. FORPS (FORth-based Production System) was developed to combine the intelligent, rule-based control of production systems with the real-time control capabilities of FORTH. It is a complete production system offering high speed, extensibility, and simplicity in a small package (object code <3K). This paper details the philosophy and design of FORPS, and presents its capabilities by way of two examples: a fast solution to the Towers of Hanoi and a simple robot obstacle avoidance program.

Introduction

A major focus in Artificial Intelligence (AI) in recent years has been the area of expert systems. An expert system is intended to capture the knowledge of an expert in a given domain and then to apply that knowledge to specific problems. Most expert systems are based in LISP, and many are designed as production systems (see below). A few systems have been written in FORTH, most notably the expert system for locomotive repair of Johnson and Bonissone [JOH83] and the more general EXPERT-2 system [PAR84]. These systems, however, are closely modeled after LISP-based production systems and as such fail to maintain the full flexibility of a FORTH-based system.

This paper presents the philosophy and design of FORPS, a FORth-based Production System. FORPS was developed as part of the Advanced Integrated Maintenance System (AIMS) at Oak Ridge National Laboratory [MAR84].[1] It was first applied in an obstacle avoidance program used by the ASM, an Advanced Servo-Manipulator tele-robotic system (described in detail in [MAT86]). FORPS was originally written in polyFORTH for the Motorola MC68000 microprocessor. Since then it has been ported to other systems including F83 on an IBM PC and C-FORTH running under UNIX[2] on a VAX 780. All code appearing in this paper is in polyFORTH[3] (79 Standard).

[1] This work was conducted under the Consolidated Fuel Reprocessing Program (CFRP) at Oak Ridge National Laboratory (ORNL) in Oak Ridge, Tennessee. ORNL is operated by Martin Marietta Energy Systems.

[2] UNIX is a registered trademark of Bell Laboratories.

[3] polyFORTH is a registered trademark of FORTH, Inc.

The intentions of this paper are to explain in greater detail the internal design of FORPS, to elaborate on its features, and to point out some of its limitations. The explanation of the design assumes that the reader is familiar with FORTH's dictionary structure and understands the purposes of the cfa, pfa, etc. To demonstrate some of its features, an example of FORPS applied to the classic Towers of Hanoi problem is provided, along with a brief description of the AIMS obstacle avoidance program. Throughout, FORPS is compared and contrasted with other production systems. A basis for this comparison is established in the following general discussion of production systems.

What is a production system?

Production systems are receiving wide attention in the area of AI as an efficient and practical means for solving well-defined yet complex problems. Many expert systems are built around production systems because they provide an effective method for coding the knowledge of experts. The distinguishing quality of a production system is that it represents knowledge in the form of "condition-action" pairs. A single pair is called a rule or "production", and is usually represented in the form: IF A THEN B. The literal meaning of such a rule is, if condition A is present or satisfied, then action B should be executed. For example, the information "punt on fourth down and long yardage"[4] might be encoded in a production rule as:

IF (it is fourth down AND there is long yardage)
THEN punt

A complete production system might have several hundred related rules [BAR81, WIN77].

In addition to its rules, a production system also needs a set of "facts" to which the rules are applied. A fact is simply an atomic piece of information. For example, "down = 4" or "down(fourth)" might represent the fact "it is forth down."

To make use of the knowledge of the rules and facts, a method is needed for drawing meaningful inferences. This is the job of the "inference engine" (IE). There are two different types of inference mechanisms commonly used: forward chaining and backward chaining. In forward chaining, facts in the data base are matched against the conditional clauses of the rules; when a match is found, the rule's action is executed. The process terminates when none of the rules can be matched to the existing facts. This type of inference mechanism is often referred to as "data driven" because it is the data that determines the application of the rules.

Backward chaining on the other hand is "goal driven." A "goal" is presented and the system attempts to meet this goal by matching it to the action portion of the rules; when a rule's action matches a goal, its conditional statement then becomes the new subgoal(s). The backward chaining process ends successfully when all goals and subgoals have been satisfied, or fails when no rules are found to be applicable to the current goal.

This paper will concentrate on forward chaining, since that is what is used in FORPS.[5] Forward chaining inference engines differ between production systems, but a simple yet standard method of rule evaluation is as follows:

1. condition scanning – mark all conditionally satisfied rules as "active"
2. rule selection – select the active rule of highest priority[6]
3. "fire" the rule – execute the "action" of the selected rule
4. repeat – repeat 1, 2 and 3 until there are no active rules

[4] Example taken from The AI Handbook, Vol. 1, p. 190. "Punt on fourth and long yardage" is a piece of common knowledge used in American football.

[5] Backward chaining can be simulated by forward chaining, so nothing is necessarily lost by this concentration on forward chaining. A demonstration on how this is accomplished is presented below in the Towers of Hanoi example.

[6] This priority can be based on many different factors such as specificity of the condition, recency of the matched information, explicit priorities assigned by the programmer, etc. See [BAR81, WIN77] for examples.

The IE repeatedly cycles through these steps until a cycle passes in which no rules were active or an explicit HALT command was executed, at which time execution of the production system terminates.

There are several advantages of production systems over conventional programming. Production rules are a convenient and efficient means for representing knowledge in a highly structured yet flexible format. Since all knowledge is represented within these well defined rules, the system maintains a high degree of homogeneity. This is desirable because it implies that all knowledge, independent of its source or specific application, can be treated similarily. Also, since IE contains most of the procedural control for the system, the individual rules can be treated as independent. This independence means new rules can be added to the system without significantly altering the action of existing rules. It is also the case that the IF...THEN structure of productions is similar to the way experts often express their knowledge—i.e. coding of expert knowledge can be facilitated with production rules.

One disadvantage of (forward chaining) production systems is that they are often slow. Part of the problem is the inefficiency of evaluating every rule's conditional statement during each cycle. The solution is to scan only a subset of the rules, such as only those rules that were active last cycle or reference facts that changed or emerged since the last cycle. This method, however, requires additional bookkeeping, adds to the time and space overhead, and slows execution when there are only a few rules in the system. Many production systems also use extensive pattern matching capabilities, and while this allows for more convenient and powerful rules it also sacrifices speed and efficiency. Furthermore, many systems use data structures that are temporary and require garbage collection schemes that can slow operations even further. As a result, traditional production systems are generally too slow to be used in the solution of time critical problems, e.g. real-time control processes.

The Philosophy of FORPS

FORPS was designed as a complete production system. It is unique in that it provides this ability from within a FORTH environment, and all its components are FORTH words. Since many FORTH applications involve real-time problems, FORPS was also designed to be as fast and efficient as possible. The design philosophy was thus three-fold: FORPS should 1) provide the representative power of a production system, 2) maintain the advantages of FORTH (including its extensibility and its interpretive nature), and 3) run at speeds appropriate for many real-time applications.

FORPS contains a set of defining words for constructing production rules and an inference engine for scanning and interpreting rules. The facts of FORPS are simply FORTH words that return boolean values (e.g. variables, constants, functions, etc.). Rule construction is comparable to standard production systems, i.e. `RULE:` (name) `*IF*` (condition) `*THEN*` (action) `*END*`. The IE is a standard forward chaining inference mechanism as described above. The syntax and semantics of the IE and the rule-defining words are described in the next sections.

The intention of maintaining a FORTH environment was for the rules to be definable by the programmer using standard FORTH words. Furthermore, it was desired that any legal FORTH word be executable at any point within a production rule. It was also seen as desirable to have the ability to use previously written FORTH code in the rules, so that intelligent systems could be made to overlay on top of existing FORTH software systems. The ultimate goal, therefore, was for an experienced FORTH programmer to be able to easily create useful production rules without first learning a large set of new commands and techniques, and while being able to use previously written FORTH code.

By allowing the execution of words throughout the rule, FORPS gains the additional feature of having conditional clauses that can contain words that perform actions.[7] In many production

[7] Some purists hold that conditions are by definition passive and thus argue that this feature encourages sloppy, unstructured rules. But for practical matters, this relaxing of structured methods seems warranted by the added flexibility.

systems the condition portion of a rule can never perform an action other than the passive pattern-matching of its conditional clause. In a real-time control situation it may be desirable to perform a quick operation—such as the polling of an I/O port—during the condition scanning phase. The ability to execute any FORTH word in either the condition phase or the action phase of a rule gives the programmer a more flexible tool.

Since FORPS was intended for use in real-time applications, it was designed to be as nearly short and simple as possible (without using code definitions).[8] Accomplishment of this goal is evident in FORPS's three screens of source code (see Appendix A). While a bare minimum system requires that some features be left out of the design, FORPS's extensibility allows modifications to be made later should they become desired. A number of potential enhancements are described below. None the less, the fundamental components of a full-fledged production system are completely subsumed within FORPS.

The Inference Engine

At the heart of any production system is its inference system (IE). The IE must accomplish three tasks as described earlier: 1) test conditionals, 2) select one rule from the set of active rules, and 3) execute the action of the chosen rule. To facilitate the processing of these tasks in FORPS, a rule-table is constructed in which the important information concerning each rule is stored (see Table 1).

RULE	C-PFA	A-PFA	Fire-cell	Priority
Rule 1	4020	4134	0	10
:	:	:	:	:
:	:	:	:	:
:	:	:	:	:
Rule n	4902	5030	-1	0

Table 1. The RULE-TABLE

The first column of the table holds a rule's conditional pfa, `C-PFA`.[9] This is a pointer to the code that is executed to determine whether the rule's conditionals are met. The second column contains the action pfa, `A-PFA`, that points to the code to be executed should the rule be chosen to be fired. The third column of the table is the location of the "active-cell" and holds the result of the conditional test for that rule, i.e. the result of executing the `C-PFA`. A true (non-zero) value in this cell indicates the rule is "active" and ready to fire. The priority of each rule—used as the basis for conflict resolution—is stored in the fourth column. This priority takes on a value between 0 and 65343 and is explicitly (and optionally) set by the programmer when the rule is defined.[10] These priorities may be altered during program execution (for example, the action of one rule may change its own or another rule's priority value).

The rule-table is created as a named (`RULE-TABLE`) table 16 bytes wide by `MAX-#RULES` long by allotting (`MAX-#RULES * RULE-LEN`) bytes of memory. `MAX-#RULES` is a constant set by the programmer to the maximum number of rules in the system. `RULE-LEN` is also a constant and

[8] Some of the time critical words in FORPS could be rewritten as code words resulting in increased execution speed, but to maintain readability and transportability in this initial version, code definitions were excluded by choice.

[9] The pfa is used rather than the cfa because polyFORTH's `@EXECUTE` operates on a word's pfa.

[10] The limit on the value of the priority is dependent on the size of the cell used to store the value, in this case a 16 bit word.

specifies the number of bytes per rule (16 was chosen rather than the minimum 14 required because this lends itself to a neat display using DUMP). Before defining a system's rules the rule-table must be cleared and the table's pointers reset, and this is accomplished with the word *RESET- FORPS*.

```
: FORPS   >RULE-TABLE @ 4- >LAST-RULE ! 0 CYCLE !
  BEGIN
        1 CYCLE +!
        CLEAR-FIRES
        TEST-RULE-CONDS
        SELECT-BEST-RULE
        FIRE-RULE
        NO-ACTIVITY @
  UNTIL ;
```

Figure 1. FORPS's Inference Engine

The convenience afforded by the rule-table becomes evident in the simplicity of the FORPS's interpreter or inference engine shown in Figure 1. The first action of the IE word FORPS is to set a pointer to the end of the rule-table and to clear the cycle counter variable CYCLE (refer to comments in Appendix A for variable and constant definitions). The inference loop is then entered, and CYCLE is incremented. CLEAR-FIRES zeros the contents of the fire cells of all the rules in the table. The conditional statements of the rules are then evaluated by TEST-RULE-COND (see Figure 2). Evaluation of the conditionals is simply a matter of executing the code pointed to by the C-PFA's. This is fast and simple, since it is accomplished with @EXECUTE within a tight DO +LOOP.[11] The effect of executing a rule's conditional clause is that the fire cell of the rule is updated with the result of the conditional's evaluation. The details of this effect will become more evident when the rule definition words are described below.

```
: CLEAR-FIRES     RT-LIMITS DO 0 I >FIRE-CELL ! RULE-LEN +LOOP ;
: TEST-RULE CONDS  RT-LIMITS DO I @EXECUTE   RULE-LEN +LOOP ;
: SELECT-BEST-RULE   NO-ACTIVITY TRUE  SET-DEFAULT
      RT-LIMITS DO I DUP >FIRE-CELL @
      IF DUP >PRIORITY W@ DUP  HIGH-PRI @ >
            IF HIGH-PRI !  >ACTION BEST-ACTIVE-RULE !
                NO-ACTIVITY FALSE
            ELSE  2DROP  THEN
      ELSE  DROP  THEN  RULE-LEN +LOOP ;
: FIRE-RULE  BEST-ACTIVE-RULE @  @EXECUTE ;
```

Figure 2. Inference engine support words

After all the conditional pfa's have been executed, each rule's active-cell will contain either a false (zero) or a true (non-zero) value. These values are used by SELECT-BEST-RULE to choose the best rule for firing. Beginning at the top of the rule-table, the active-cell of each rule is tested in turn for a true value indicating the rule is active and ready for firing. The first active rule found becomes the "best-active-rule" and its A-PFA is stored in BEST-ACTIVE-RULE.

[11] Again this could be made more efficient with code definitions at the cost of readability and transportability.

Since additional rules may also be active, the rest of the active-cells must be tested for true values. Each time another active rule is encountered, "conflict-resolution," based on rule priority, is used to select the best rule. As SELECT-BEST-RULE scans down the active-cell column and finds an active rule, it fetches that rule's priority value and compares it with the priority level of the current best-active-rule stored in HIGH-PRI. If this rule's priority is higher than the currently selected best-active-rule, it takes that rule's place as the best-active-rule. After the active-cells of all the rules in the table have been scanned, BEST-ACTIVE-RULE contains the A-PFA of the rule chosen to fire.

FIRE-RULE fires the best-active-rule by performing a @EXECUTE on the contents of BEST-ACTIVE-RULE. In the event that no rules are active, BEST-ACTIVE-RULE points to NOOP that was set as its default by SET-DEFAULT.

The boolean variable NO-ACTIVITY will be true if all the fire cells contained false values or if the word HALT was executed by the fired rule (HALT is defined as NO-ACTIVITY TRUE). NO-ACTIVITY supplies the conditional termination flag for the BEGIN-UNTIL loop of the inference engine. In other words, the inference engine continues until there is NO-ACTIVITY.

The main advantage of the rule-table is the efficiency realized in the IE. The loops for testing conditionals and evaluating active rules are tight and constitute near minimal overhead.[12] For the most part these loops simply fetch or execute the contents of memory locations. The reason the rule-table is able to be so efficient is that most of the computational work is accomplished in the creation of the table. This removes the cpu intensive operations from the time critical inference loop. The next section describes how the rule-table is constructed using FORPS's rule defining words.

Rule definition

Rules in FORPS are defined much in the same way that words are defined in FORTH. Five words are used in the definition of rules: RULE:, PRIORITY:, *IF*, *THEN* and *END*. A rule is defined first by giving it a name with RULE: followed by an optional priority specification with PRIORITY:. The conditional clause is provided between the words *IF* and *THEN*, and the action statement occurs between *THEN* and *END*. The effect of defining a rule is to add its name, its conditional parameter field, and its action parameter field to the normal FORTH dictionary, and to add a row to the rule-table containing pointers back to the rule's dictionary entry (see Fig. 3).

To understand how these words function consider the following example:

```
RULE: FOURTH-&-LONG     PRIORITY: 1
 *IF* DOWN 4 =
      YARDAGE LONG >
 *THEN* PUNT
 *END*
```

The name of this rule is FOURTH-&-LONG and it has a priority level of 1 (since priorities are relative, this priority has meaning only after other rules are added). It represents the knowledge that if DOWN is equal to 4 and YARDAGE is greater than LONG, then PUNT (the example given earlier). Since all words used in the rule's definition must exist in the FORTH dictionary, the words DOWN, YARDAGE, LONG and PUNT must be predefined FORTH words. Presumably, in this example, DOWN will return a value between 1 and 4, YARDAGE will return the number of yards needed for a first down, LONG is a value for long yardage, and PUNT specifies the action to be carried out to perform a punt.

[12] On each cycle the IE performs to passes through the rule-table. It would be possible to reduce this to one pass, thereby improving speed and reducing space (i.e. the active-cell would no longer be necessary). These benefits would come at the cost of reduced flexibility in terms of future enhancements that might make use of the active-cell contents (see below).

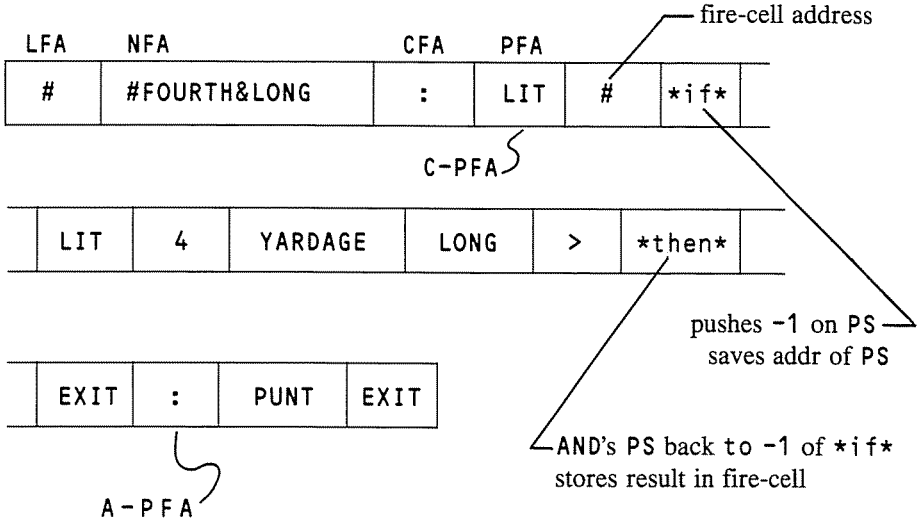

Figure 3. Rule Dictionary Entry

RULE: is similar to the standard FORTH word : (see Fig. 4). It creates a normal colon definition in the dictionary with the name FOURTH-&-LONG. In addition, it adds an entry to the rule-table (after testing for sufficient room). The pfa of this word is stored in the first column of the row of the rule-table pointed to by >RULE-TABLE. This address is a pointer to the code to be executed by TEST-RULE-COND in calculating the value of the conditional statement—i.e. it is the C-PFA. RULE: terminates by putting the FORTH interpreter into compile mode to begin compiling the conditional code into the dictionary.

```
: RULE:   >RULE-TABLE RULE-LEN / MAX-RULES = ABORT" no room"
      CURRENT W@ CONTEXT W!  CREATE  COND-PFA!  -4 ALLOT
      COLON-CFA ,  SMUDGE | ;
: PRIORITY:   >RULE-TABLE @ -' IF NUMBER ELSE DROP EXECUTE
      THEN SWAP >PRIORITY W! ; IMMEDIATE
: *if*   -1 'S 4- 'SP-IF ! ;
: *IF*    >RULE-TABLE @ >FIRE-CELL [COMPILE] LITERAL
      COMPILE *if* ;  IMMEDIATE
: *then*  ( *n) -1 'SP-IF @ 'S DO AND 4 +LOOP  SWAP ! ;
: *THEN*   COMPILE *then* COMPILE EXIT COLON-CFA ,
     ACTION-PFA! ;  IMMEDIATE
: *END*   RULE-LEN >RULE-TABLE +!  COMPILE EXIT  SMUDGE
     R> DROP ;  IMMEDIATE
```

Figure 4. Rule defining words

PRIORITY: is an immediate word. Its effect is to store the value of the next word in the input stream into the current rule's priority cell within the rule-table.[13] Any executable word that returns a value on the stack may appear after PRIORITY: – typically, however, this word is a number. Inclusion of the PRIORITY: is optional; if omitted its default value is zero.

The FORPS word *IF* is also immediate. It first calculates the address of the current rule's active-cell within the rule-table (>RULE-TABLE @ >FIRE-CELL). This address is compiled into the conditional definition as a literal so that at run-time (i.e. when the conditional clause is tested) it will be pushed onto the parameter stack as the address where the conditional result will be stored. *if* is then compiled into the definition; it will also be executed by TEST-RULE-COND when the rule's condition is evaluated. At run-time, *if* marks the current top of the parameter stack and pushes a −1 onto it as a flag for later use by *then* (see next paragraph). All words following the *IF* up until *THEN* are compiled straight into the dictionary just as in the definition of a normal word—this is the reason all rule components must be predefined FORTH words.

THEN is another immediate word and compiles *then*, EXIT, and the cfa of : into the definition of the rule. The word *then* will be executed each time the rule's conditional clause is tested; it has the effect of ANDing everything on the parameter stack back to the −1 stack element put on the stack by the execution of *if*. This ANDing converts the elements left on the stack by the words in the conditional clause into a single boolean value. After determining this value, *then* stores it into the rule's active-cell, the address of which was put on the stack at the beginning of the conditional clause. *THEN* also compiles an EXIT immediately after the *then* so that execution of the conditional pfa will terminate after loading the active-cell. The cfa of : is then compiled into the definition to mark the beginning of the action code. The address of the next dictionary cell is loaded into the second column of the rule-table as the A-PFA, since this will be the start of the action clause. All words following the *THEN* are compiled directly into the dictionary just as in standard word compilation.

The word *END* is identical in function to ; except it also increments the rule-table pointer that points to where the next rule will be entered in the table.

The dictionary entry created for the rule can be thought of as two words possessing a common head, i.e. the conditional word and the action word (Fig. 3). The rule's head (name) is never specifically used by FORPS, but it is useful for debugging purposes (executing the rule outside of the IE by invoking its name, effectively loads that rule's active-cell with the value of the conditional clause). It could also be useful in an enhanced FORPS system to possibly display the name of each rule as it fires or print a list of the last few rules to fire (see enhancements below).

A word appearing in the condition or action clause of a rule can be any predefined FORTH word. Some concern, however, must be given to the way FORPS processes the conditional clause. Words in the conditional clause need not leave anything on the stack; if nothing is left on the stack the rule will always be active as a result of the ANDing of the two −1's put on the stack by *if* and *then*. Boolean values left on the stack should be either −1 for true or zero for false. There may be times, however, when it is desirable to use other values for true.[14] In such an event, care must be taken since it is not necessarily the case that several true (non-zero) values will AND to a true value. For example if 5 and 2 were left on the stack by the conditional words, the ANDing of these values with the two −1's would result in a value of zero being stored in the rule's active-cell.[15]

[13] Interpreting the next word in the input stream is more complicated than if the value was simply put on the stack before PRIORITY:, but the former lends itself to more readable rules.

[14] One case would be if the conditional clause needed to pass a value to the action statement. The active-cell could then be used for parameter passing, with the action code locating the active-cell through an address compiled in its definition.

[15] Conservative FORTH programmers may object to the implicit ANDing of the stack elements in this manner. This mechanism was included in order to conform with the standard production rule convention of implicit ANDing of conditional clauses, but it could be removed without loss of generality. The gain would be evident in improved execution times, at a cost of putting greater responsibility of stack maintenance on the programmer.

Example 1: Towers of Hanoi

The Towers of Hanoi problem has been programmed in FORPS; the four screen source listing appears in Appendix B. The Towers of Hanoi is a classic AI problem and was selected to demonstrate FORPS's capabilities in this area. The rules will be explained on a high level and the reader is referred to [DWE85] for further details of the algorithm used.

Many forward chaining production systems make use of "goals." A goal is some condition that needs to be met to solve the problem at hand. Rules in the production system are designed to look for and resolve specific goals, and in the process may spawn other sub-goals. Goals in effect give the inference engine some "direction" as to which rules are to be applied and in what order; technically speaking this method simulates the effects of backward chaining.

The "goalstack" in the FORPS solution to the Towers of Hanoi is used to store goals and sub-goals. It is a standard last-in, first-out stack operating on 16 byte goal elements. At any one time, only a single goal—the one at the top of the goal stack—is active. Rules are not allowed to look at any goal other than the currently active goal, although they may freely add new goals to the top of the stack. With this convention, the system is forced to focus its attention on one goal at a time.

Each goal consists of four values: the type of move (`MOVE-TOWER` or `MOVE-DISK`), the number of disks to move, the `SOURCE` peg, and the `TARGET` peg. Several words were written to facilitate use of the stack within the productions. `ADDGOAL` and `REMOVEGOAL` add and remove goals from the stack respectively. `IS-GOAL` and `IS-#-OF-DISKS` are used to determine the type of goal and the number of disks in the current goal. For example, `MOVE-TOWER IS-GOAL` will return true if the current goal is to `MOVE-TOWER`.

`START-TOWERS` is the first rule; it fires once at the beginning to initialize the goalstack with a null goal and to submit the first goal: `MOVE-TOWER` of height `#DISKS` from the `HOME` peg to the `GOAL` peg. The next rule is the `MOVE-TOWER-RULE`, which is active whenever the top goal is to `MOVE-TOWER`. It replaces the current goal with two `MOVE-TOWER` goals and a `MOVE-DISK` goal.

`MOVE-SINGLE-DISK-TOWER` is the rule that fires when the top goal is a `MOVE-TOWER` goal and the number of disks to move is one. This rule has a higher priority than the `MOVE-TOWER-RULE` because it is possible for both to be active at the same time, in which case the former should fire. The rule `MOVE-SINGLE-DISK` fires when the current goal is `MOVE-DISK`. Its effect is to remove the current goal and print a message to move a disk from the source peg to the target peg.

A high level word, `TOWERS`, is used as a user interface to the production system. `TOWERS` prompts the user for the number of disks to move from the source to the target, and sets `STARTED` to false. When FORPS is then called, the first rule matches to the value of `STARTED` and the system begins solving the problem. When the problem is completed the top goal will be the null goal, which will not match any of the rules, and as a result FORPS will terminate.

This program was run on a 68000 microprocessor operating at 10 MHz on problems with difficulty varying from five to ten disks. The results are listed below in Table 2. Included in this table are the number of rules fired for each run alongside the results from a five rule solution used to measure the minimum overhead of a rule. In these tests all I/O operations were removed so that the results could potentially be compared with other systems and machines.

Solution Times (seconds)			
# Disks	4 rules	5 rules	rules fired
5	.29	.33	64
6	.58	.66	128
7	1.15	1.33	256
8	2.31	2.66	512
9	4.63	5.31	1024
10	9.25	10.62	2048

Table 2. Towers of Hanoi Results

The average time required for each inference made during the solution of a given problem can be calculated by dividing the total time for the solution by the number of rules fired. This works out to an overhead of approximately 4.5 ms per rule, or a little over 220 inferences per second. For other problems, the overhead will vary depending on the number of rules, the complexity of the conditional clauses, and the demands of the action statements.

The minimum overhead for any rule in any system is determinable by comparing the differences in execution times between the four and five rule systems (Table 2). The extra rule added to the five rule solution has the minimum conditional clause (the constant zero) which is always false. Since this rule will never be active but will be scanned each cycle, the difference in time between this solution and the four rule solution, divided by the number of cycles, reveals the minimum time requirement for a single rule. The value obtained is roughly .7 ms, which implies that each rule added to a system will consume at least .7 ms per cycle.

Timing results were not available for this problem running on the same machine under different production systems. Results, however, were obtained for OPS5[16] running on a number of different machines (e.g. DEC10, VAX 780). While a clear comparison cannot be made on these times, it is somewhat significant that the fastest realized time on any machine was 28 seconds for the seven disk problem.

It is not here implied that a value judgement can be made between the systems on the basis of this one example. This problem uses a small number of rules, and reveals nothing about how FORPS will perform on more complex tasks. Since each rule in a FORPS system will require at least .7 ms per cycle, a system with more than 700 hundred rules would require at least .5 seconds per inference. OPS5, on the other hand, does not test each rule every cycle—only those conditionals dependent on memory that changed since the last cycle are tested. With a very large rule base, OPS5's limited scanning schemes would probably excel. There are, however, ways of limiting the number of rules scanned by FORPS as well (see below.)

Another reason for FORPS's dramatic showing in these tests is in its use of the goalstack data structure, which is much simpler than the structures employed by OPS5 and similar systems. This possible criticism, however, highlights one of the features of FORPS: the ability to create and manipulate unique data structures from within rules.

Example 2: A simple obstacle avoidance system

A more practical application of FORPS is the real-time problem of obstacle avoidance. The ASM tele-robot at ORNL is transported through its cell environment by way of an overhead transport system. An operator controls the transporter with a joystick while observing the robot's motion through several cameras and with the aid of a computer generated cell map display. The cell map shows the manipulator's position as well as the location of potential obstacles within the cell. Information concerning the height and location of potential obstacles is statically maintained within the cell map's data base.

A simple production system consisting of seven rules has been written to assist the operator in obstacle avoidance during manipulator transportation (see [MAT86] for details). This obstacle avoidance system sits on top of the existing transporter control software. Its function is to analyze the operator's direction requests and prevent the execution of commands which would run the manipulator into an obstacle. The production system scans the data stored in the cell map data base to decide if the manipulator is being directed towards an object of potentially dangerous height. If the manipulator is being directed into an obstacle, a warning is displayed, but the operator is allowed to proceed in the desired direction. If the object is an absolute obstacle (i.e. the manipulator would make contact with the object), the operator is not allowed to continue in the requested direction;

[16] OPS5 is a popular production system written in LISP and frequently used in the development of expert systems.

instead, an obstruction message is displayed and the transporter is halted. In certain instances, however, when it is necessary to proceed cautiously into an obstacle, an "over-ride" switch may be used to overrule the system's decision.

What is significant in this example is that the obstacle avoidance system was written using existing FORTH code and data structures. Words to access and display the cell map's data and to move the transporter had already been written and were being used in an unintelligent cell-map/ transporter system. Because of the existence of these words, the design, implementation, and testing of the obstacle avoidance production system was accomplished in a short time—roughly four man-hours. Furthermore, this simple system can be easily extended by the addition of further rules to give it greater intelligence. One enhancement being considered is the addition of an automated path-finding routine, in which the operator specifies a destination and the program autonomously transports the manipulator safely through the cell to that location.

Extensions to FORPS

Because FORPS was designed as a minimum system in order to make it as fast and simple as possible, many features were left out. In this section, the implementation of a few of these features will be discussed to give a flavor of the extensibility and versatility of FORPS.

One criticism against FORPS is that the IE executes the `C-PFA` of every rule each cycle. In a system composed of many rules this expense would likely make the code too slow for real-time application. Many popular production-based systems use some type of "recency control" to test only those rules accessing the more recent facts. In FORPS, however, this approach would greatly restrict the flexibility of the types of facts allowed and preclude "facts" in the form of words that perform calculations on the spot. Another solution to this problem looks more promising. Almost any problem can be divided into several sub-components, with rules specific to those components. It would be a simple matter to construct multiple rule-tables for these different components and switch between the tables by having high-level rules change the `RULE-TABLE` and `>LAST-RULE` pointers. In this way a minimum number of rules would be scanned each cycle while maintaining the potential for a large rule base.

FORPS's conflict resolution procedure might also be enhanced. The priority criteria currently used in FORPS for conflict resolution is very simple. Other priority systems exist that test, for example, the recency of the facts, the number of cycles each rule has been active, and/or the specificity of the conditional matches. Implementing the recency test would restrict FORPS as mentioned above. Saving the active durations of rules, however, could be simply accomplished by adding another column to the rule-table and extending the `SELECT-BEST-RULE` process to increment this cell if the rule is active and to clear it if inactive. The criteria for selecting the highest priority rules would have to be changed so as to become a function of this new activity value. A simple measure of specificity (for example, the number of words) of each conditional clause could be added as another column of the rule-table, and used in the event that two rules of equal priority were active. The merit of these types of modifications, however, is debatable—while modeling other production systems, they detract from the simplicity of FORPS.

Another potential alteration to the priority scheme, more in the spirit of FORPS, is to eliminate the priority value altogether. Instead of a specified value, priority would be determined by the ordering of the rules. The first active rule encountered by `BEST-ACTIVE-RULE` would be selected and fired. In a large rule base, this "enhancement" might greatly improve execution speeds. The cost, however, would be a reduction in programming flexibility since the rules would need to be explicitly ordered. An alternative to discarding the priority altogether is to keep it for programming convenience but use it at compilation time to sort the rows in the rule-table according to relative priorities.

An important component of many expert systems is the ability of the system to explain to the user how or why a conclusion was drawn. A simplistic approach to this idea in FORPS might list a history of the most recent rules that fired. A question of why a rule conclusion was made could then be answered with an explanation such as "because rule X fired," which in turn could be explained "because rule Y fired." This method might be implemented in FORPS by creating a ring buffer into which the address of a rule's name is stored when it fires. Depending on the size of the buffer, the system could store an arbitrarily long history of fired rules. To explain a conclusion, the most recent address could be fetched and its name displayed. While this would not provide an extensive explanation, it would be helpful for system development and could be made more extensive (e.g. supplement the rules with detailed explanations to be displayed instead of the names).

Two enhancements used during the development of the aforementioned examples were "trace" and "single step" features. As a rule fired, its name was displayed and execution was halted until a key was pressed by the operator. This was implemented simply by adding a word to `FIRE-RULE`. This word used the address in `BEST-ACTIVE-RULE` to find the name of the rule just fired, after which it displayed the name and waited for a key to be pressed. A more extensive development tool could be created in a similar manner that might include features to begin and end the IE on any rule and to alter the contents of variables before proceeding.

Summary

As FORTH is applied to increasingly more difficult problems, the need for intelligent control is becoming accentuated. Intelligent methods of programming are being developed in AI, and many are becoming applicable as practical solutions to real world problems. It is both desirable and necessary to complement FORTH with AI programming—this is the intention of FORPS. FORPS integrates the AI programming technique of production systems with the many features of a FORTH environment, to afford effective and more intelligent solutions to difficult problems.

FORPS is a complete production system, possessing an inference engine and tools for creating production rules. It has been shown through two examples, the Towers of Hanoi problem and an obstacle avoidance task, to be capable of both fast computation and practical real-time control. As noted, it was designed for speed and simplicity, and as a result does not possess all the nice features of slower, more extensive systems. What FORPS does possess, however, is extensibility—i.e. these features can be added if necessary.

A unique quality of FORPS is that it exists within a complete FORTH environment; it is written in FORTH and the components of the rules created by the programmer are all FORTH words. Consequently, it is a system that should be easy to learn for FORTH programmers with little or no prior knowledge of production systems. Experienced FORTH programmers should find FORPS useful for adding "intelligent" control to existing FORTH systems.

Acknowledgements

I would like to thank Bill Hamel and Lee Martin for the direction and support they gave to this work. I am also grateful to Richard Spille, Steven Killough, Bill Dress and the reviewers for their insightful comments and criticism.

References

[BAR81] Barr, A. and Feigenbaum, E.A. *The Handbook of Artificial Intelligence*. Vol. 1. Los Altos, CA: William Kaufmann, Inc., 1982.

[DWE85] Dwedney, A.K. "Computer Recreations." *Scientific American* 251 (November 1984).

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

[MAR84] Martin, H. L., Hamel, W.R., Killough, S.M., and Spille, R.F. "Control and Electronic Subsystems for the Advanced Servomanipulator." In the proceedings of the American Nuclear Society Topical Meeting on Robotics and Remote Handling in Hostile Environments. Gatlinburg, TN. April 1984.

[MAT86] Matheus, C.J. and Martin, H.L. "FORPS: a FORTH-based Production System and its Application to a Real-time Control Problem." Paper presented at the ASME Computers in Engineering Conference. Chicago. July 1986.

[PAR84] Park, J. "Expert Systems and the Weather." *Dr. Dobb's Journal* (April 1984): 24-31.

[WIN77] Winston, P.H. *Artificial Intelligence*. Addison Wesley, 1977.

Christopher J. Matheus received a B.A. in physics in 1983 from Lawrence University, Appleton, Wisconsin. For a year he traveled abroad as a Thomas J. Watson Fellow studying "The Impact of Industrial Robots in Japan and Great Britain." The past three summers he has worked at Oak Ridge National Laboratory in Oak Ridge, Tennessee, as a technical assistant programming in FORTH on advanced tele-robotic systems and artificial intelligence applications. Currently, he is a graduate student doing research in Machine Learning at the Artificial Intelligence Laboratory of the University of Illinois at Urbana-Champaign, while working towards a Ph.D. in Computer Science.

Manuscript received December 1985.

Appendix A

```
460 LIST
 0 ( FORPS -- a FORth-based Production System )      ( CJM 8/19/85)
 1 1 3 +THRU
 2 EXIT
 3
 4 FORPS is under the copywrite of Martin Marietta Energy Systems.
 5       It is in the public domain and may be used freely
 6       as long as no false claims are made to its authorship.
 7       Author:   Christopher J. Matheus
 8                 University of Illinois
 9                 222 Digital Computer Lab
10                 1304 W. Springfield Ave.
11                 Urbana, IL   61801
12
13 This software was developed at Oak Ridge National Laboratory,
14 Oak Ridge, TN, under the Consolidated Fuel Reprocessing Program.
15

461 LIST
 0    ( FORPS constants and variables )              ( CJM*8/15/85)
 1 10 CONSTANT MAX-#RULES   16 CONSTANT RULE-LEN
 2 VARIABLE NO-ACTIVITY   VARIABLE 'SP-IF        VARIABLE 'NOOP
 3 VARIABLE >RULE-TABLE   VARIABLE >LAST-RULE    VARIABLE CYCLE
 4 VARIABLE HIGH-PRI      VARIABLE BEST-ACTIVE-RULE
 5 CREATE RULE-TABLE      MAX-#RULES RULE-LEN * ALLOT
 6
 7 : >ACTION   ( a -a)  4 + ;
 8 : >FIRE-CELL   ( a -a)  8 + ;
 9 : >PRIORITY   ( a -a)  12 + ;
10 : HALT   NO-ACTIVITY TRUE ;
11 : *ERROR*   1 ABORT" NO RULES LOADED" ;
12 : *RESET-FORPS*   RULE-TABLE DUP >RULE-TABLE !  MAX-#RULES
13    RULE-LEN * ERASE  [']  *ERROR* RULE-TABLE ! ;  *RESET-FORPS*
14 : NOOP ;   ' NOOP 'NOOP !   ' NOOP 4- @ CONSTANT COLON-CFA
15

462 LIST
 0    ( FORPS rule defining words )                  ( CJM*8/15/85)
 1 : COND-PFA!      HERE >RULE-TABLE @ ! ;
 2 : ACTION-PFA!      HERE >RULE-TABLE @ >ACTION ! ;
 3 : RULE:   >RULE-TABLE RULE-LEN / MAX-#RULES = ABORT" no room"
 4    CURRENT W@ CONTEXT W!  CREATE  COND-PFA!  -4 ALLOT
 5    COLON-CFA ,   SMUDGE ] ;
 6 : PRIORITY:   >RULE-TABLE @  -' IF NUMBER ELSE DROP EXECUTE THEN
 7    SWAP >PRIORITY W! ;  IMMEDIATE
 8 : *if*   -1 'S 4- 'SP-IF ! ;
 9 : *IF*   >RULE-TABLE @ >FIRE-CELL [COMPILE] LITERAL
10    COMPILE *if* ; IMMEDIATE
11 : *then*   ( *n)  -1  'SP-IF @ 'S DO AND 4 +LOOP  SWAP ! ;
12 : *THEN*   COMPILE *then* COMPILE EXIT  COLON-CFA ,
13    ACTION-PFA!  ; IMMEDIATE
14 : *END*   RULE-LEN >RULE-TABLE +!  COMPILE EXIT  SMUDGE
15    R> DROP ;   IMMEDIATE
```

```
1060 LIST

 0              FORPS -- FORth-based Production System
 1   FORPS is a simple Production System designed to take full
 2 advantage of the powers of FORTH.  Basic production-like rules
 3 are constructed using the words RULE:, *IF*, *THEN* and *END*.
 4 These rules are put into a table of the following format:
 5        COND. pfa  /  ACTION pfa  /  FIRE cell  /  PRIORITY value
 6        four bytes /  four bytes  /  four bytes /  two bytes
 7 Rule 1      *            *             *          *
 8    :        :            :             :      :
 9 Rule n      *            *             *          *
10    The COND. pfa is the pfa of the conditional portion of the
11 rule and the ACTION pfa is the action portion's pfa.  The FIRE
12 cell holds the result of the conditional pfa's execution.
13 PRIORITY is a number between 0 and 65534 used in the choosing
14 of the best-active-rule during 'conflict-resolution'.
15

1061 LIST
 0 FORPS constants and variables
 1 MAX-#RULES maximum number of rules allowed in table
 2 RULE-LEN length of a single rule table entry
 3 NO-ACTIVITY true if no rules fired during the cycle
 4 'SP-IF saves the parm. stack addr. at start of cond.
 5 >RULE-TABLE pointer into the rule table
 6 >LAST-RULE address of last rule in table
 7 CYCLE number of cycles executed
 8 HIGH-PRI highest priority of all rules fired
 9 BEST-ACTIVE-RULE action pfa of highest priority active rule
10 >ACTION, >FIRE FLAG, >PRIORITY offsets into rule-table
11 HALT sets NO_ACTIVITY to true -- causes FORPS to terminate
12 *ERROR* stored as 1st rule when RESET - aborts from FORPS
13 *RESET-FORPS* clears rule-table and loads *ERROR* as 1st rule
14 NOOP no-operation, used as default BEST-ACTIVE-RULE
15 COLON-CFA cfa of : -- the contents of the cfa of : words

1062 LIST
 0 FORPS rule words
 1 COND-PFA!  stores condition-pfa of rule into table
 2 ACTION-PFA!  stores action-pfa of rule into table
 3 RULE:  adds a rule to the dictionary -- creates two words
 4   with one head: first is cond word, second is action word
 5 PRIORITY:   loads the rule's priority cell with the value of
 6   the next word in the input stream (usually a number)
 7 *if*   runtime version of *IF*, saves stack pointer and puts
 8  a -1 on stack for subsequent use by *then*
 9 *IF*   compiles as a literal a pointer to the rule's
10   fire-cell (for use by *then*), and compiles *if*
11 *then*  runtime version of *THEN* -- AND's the cond. stack
12   items and stores the result in the rule's FIRE cell
13 *THEN*  compiles *then*, compiles EXIT to stop cond. word,
14   and compiles COLON-CFA to begin rule's action word
15 *END*  incrms. rule counter and ends rule compilation
```

```
463 LIST
 0    ( FORPS Inference engine )                          ( CJM*8/15/85)
 1 : SET-DEFAULT   -1 HIGH-PRI !  'NOOP BEST-ACTIVE-RULE ! ;
 2 : RT-LIMITS   ( -n n)  >LAST-RULE @  RULE-TABLE ;
 3 : CLEAR-FIRES   RT-LIMITS DO 0 I >FIRE-CELL !  RULE-LEN +LOOP ;
 4 : TEST-RULE-CONDS   RT-LIMITS DO I @EXECUTE  RULE-LEN +LOOP ;
 5 : SELECT-BEST-RULE   NO-ACTIVITY TRUE  SET-DEFAULT
 6    RT-LIMITS DO I DUP >FIRE-CELL @
 7                IF DUP >PRIORITY W@ DUP HIGH-PRI @ >
 8                   IF HIGH-PRI !  >ACTION BEST-ACTIVE-RULE !
 9                      NO-ACTIVITY FALSE
10                   ELSE 2DROP THEN
11                ELSE DROP THEN RULE-LEN +LOOP ;
12 : FIRE-RULE   BEST-ACTIVE-RULE @  @EXECUTE ;
13 : FORPS   >RULE-TABLE @ 4- >LAST-RULE !  0 CYCLE !
14    BEGIN 1 CYCLE +!  CLEAR-FIRES  TEST-RULE-CONDS
15          SELECT-BEST-RULE  FIRE-RULE  NO-ACTIVITY @ UNTIL ;
```

Appendix B

```
480 LIST
 0 ( TOWERS OF HANOI IN FORPS )                          ( CJM*8/21/85)
 1 ( FORPS must be loaded )
 2 1 6 +THRU
 3
 4 EXIT
 5    Benchmarks on 68000 10MHz cpu under polyFORTH (without I/O)
 6      #Disks         Time (sec)
 7        3              .07            Chris J. Matheus
 8        4              .14            August 15, 1985
 9        5              .29
10        6              .58
11        7             1.15
12        8             2.31
13        9             4.63
14       10             9.25
15
```

```
481 LIST
 0    ( Constants and variables )                        ( CJM*8/15/85)
 1  10 CONSTANT MAX-#DISKS
 2   1 CONSTANT HOME                2 CONSTANT GOAL
 3   1 CONSTANT MOVE-TOWER          2 CONSTANT MOVE-DISK
 4
 5 CREATE GOALSTACK   MAX-#DISKS 1- DUP * 2+ 16 * ALLOT
 6 VARIABLE GS.PTR    GOALSTACK GS.PTR !
 7
 8 VARIABLE SOURCE    VARIABLE TARGET   VARIABLE SPARE
 9 VARIABLE #DISKS    VARIABLE STARTED
10
11
12
13
14
15
```

```
1063 LIST
  0 Inference engine
  1 SET-DEFAULT sets BEST-ACTIVE-RULE to NOOP, clears priority
  2 RT-LIMITS  puts the rule-table's addr. limits on the stack
  3 CLEAR-FIRES clears the fire flags of all rules
  4 TEST-RULE-CONDS executes in order each rule's condition pfa
  5 EXECUTE-ACTIVE-RULES executes the action of the highest
  6   priority rule which has fired.  If no rules fired NO-ACTIVITY
  7   will be set to true and the FORPS loop will terminate
  8 FIRE-RULE  fires the BEST-ACTIVE-RULE
  9 FORPS  the main inference loop of the production system.
 10   After resetting #RULES and CYCLES, the inference loop is
 11   entered and continues until a cycle passes in which no
 12   rules have fired.  The inference loop has three functions: it
 13   increments the cycle counter, clears all fire cells, tests
 14   the conditional clauses,  and executes the highest priority,
 15   active rule.


1080 LIST
  0 The Towers of Hanoi in FORPS
  1
  2    The Towers of Hanoi is a classic AI problem and is presented
  3 here as an example of FORPS's potential to function like a typical
  4 production system.  It also serves to demonstrate the lightning-
  5 fast fast speed of FORPS.  The benchmarks given compare
  6 favorably to an OPS5 system running on a DEC-10 solving the
  7 problem with 7 disks in 28 seconds (1.15 for FORPS).
  8
  9    For further information concerning the algorithm used in the
 10 solution of the Towers of Hanoi problem refer to A.K. Dewdney's
 11 'Computer Recreations' column in the Nov '84 issue of Scientific
 12 American.
 13
 14
 15


1081 LIST
  0 Towers of Hanoi: constants and variables
  1
  2 MAX-#DISKS  the maximum number of disks allowed
  3 HOME the original home peg
  4 GOAL the original goal peg
  5 MOVE-ENTER the code for a tower move
  6 MOVE-DISK the code for a disk move
  7 GOALSTACK the beginning of the goalstack (see next shadow)
  8   its size is dependent on the max. number of disks allowed
  9 GS.PTR a pointer to the current top of the goalstack
 10 SOURCE the last goal's source peg
 11 TARGET the last goal's target peg
 12 SPARE the last goal's spare peg
 13 #DISKS the last goal's number of disks
 14 STARTED a flag to indicate that the problem has begun
 15
```

```
482 LIST
 0    ( GOALSTACK support words )                    ( CJM*8/15/85)
 1 : >GSTACK   ( n n - n)  DUP ROT SWAP ! 4+ ;
 2 : GSTACK>   ( n - n n)  4- DUP @ ;
 3
 4 : SPARE!   (-)  SOURCE @  TARGET @  OR 7 XOR SPARE ! ;
 5 : ADDGOAL   ( n n n n)  GS.PTR @  >GSTACK >GSTACK >GSTACK
 6    >GSTACK GS.PTR ! ;
 7 : REMOVEGOAL   ( - )  GS.PTR @  GSTACK> DROP  GSTACK> #DISKS !
 8    GSTACK> SOURCE !  GSTACK> TARGET !  SPARE!  GS.PTR ! ;
 9
10 : IS-GOAL   ( n)  GS.PTR @  4 - @ = ;
11 : IS-#-OF-DISKS   ( n)  GS.PTR @ 8 - @ = ;
12
13 : .PEG   ( n)  DUP 1 = IF ." A" DROP ELSE 2 = IF ." B"  ELSE
14    ." C" THEN THEN ;
15

483 LIST
 0    ( Start towers rule )                          ( CJM*8/15/85)
 1
 2 *RESET-FORPS*
 3
 4 RULE: START-TOWERS   PRIORITY: 10
 5       *IF*   STARTED @ NOT
 6       *THEN* GOALSTACK GS.PTR !
 7              Ø          Ø       Ø    Ø     ADDGOAL
 8              MOVE-TOWER  #DISKS @  HOME  GOAL  ADDGOAL
 9              STARTED TRUE
10       *END*
11
12
13
14
15

484 LIST
 0    ( Move tower rules )                           ( CJM*8/15/85)
 1
 2 RULE: MOVE-TOWER-RULE
 3       *IF*   MOVE-TOWER IS-GOAL
 4       *THEN* REMOVEGOAL
 5              MOVE-TOWER  #DISKS @ 1-  SPARE @  TARGET @  ADDGOAL
 6              MOVE-DISK   #DISKS @     SOURCE @ TARGET @  ADDGOAL
 7              MOVE-TOWER  #DISKS @ 1-  SOURCE @ SPARE @   ADDGOAL
 8       *END*
 9
10 RULE: MOVE-SINGLE-DISK-TOWER        PRIORITY: 1
11       *IF*   MOVE-TOWER IS-GOAL  AND  1 IS-#-OF-DISKS
12       *THEN* REMOVEGOAL
13              MOVE-DISK  1  SOURCE @  TARGET @  ADDGOAL
14       *END*
15
```

```
1082 LIST
 Ø Towers of Hanoi:  the goalstack
 1    The goalstack is a normal last-in, first-out stack of items
 2 called goals.  Each goal consists of a target peg, a source peg,
 3 the number of disks on the source peg, and the type of move.
 4 The stack structure is as follows:
 5             TARGET   SOURCE   #DISKS   TYPE-OF-MOVE
 6 goal 1       (peg)    (peg)     n         (move)
 7   :            :        :       :           :
 8 goal n       (peg)    (peg)     n         (move)
 9    The top goal is the only goal that the rules in the PS every
10 look at.  Goals are added to the stack with ADDGOAL and removed
11 from the stack with REMOVEGOAL.
12
13 IS-GOAL references the goal type of the last goal to be removed
14 IS-#-OF-DISKS references the # of disks of the last goal
15 .PEG prints A, B, or C depending on the peg number (1,2, or 4)

1083 LIST
 Ø Towers of Hanoi:  the rules
 1
 2    The first rule initializes the problem by making a goal to
 3 move a tower of #DISKS size from the HOME peg to the GOAL peg.
 4 The variable STARTED is set to TRUE which has the effect of
 5 preventing START-TOWERS from every firing again.
 6 (The first goal of Ø Ø Ø Ø simply serves to mark the bottom of
 7  goal stack.)
 8
 9
10
11
12
13
14
15

1084 LIST
 Ø Towers of Hanoi:  the rules
 1
 2    MOVE-TOWERS recognizes a goal to move a tower and replaces
 3 this goal with three new goals (see Dwedney for explanation).
 4 MOVE-SINGLE-DISK-TOWER matches a goal to move a tower where the
 5 number of disks is one.  It replaces this goal with a goal to
 6 move a single disk from the source to the target peg.
 7
 8
 9
10
11
12
13
14
15
```

```
485 LIST
 0    ( Move single disk )                                ( CJM*8/15/85)
 1
 2 RULE: MOVE-SINGLE-DISK
 3       *IF*   MOVE-DISK IS-GOAL
 4       *THEN* REMOVEGOAL
 5              TARGET @ SOURCE @
 6              ."   Move disk on peg " .PEG ."  to peg " .PEG   CR
 7
 8       *END*
 9
10
11
12
13
14
15

486 LIST
 0    ( High level TOWERS word )                          ( CJM*8/15/85)
 1
 2 : .HEADER   CR CR ." TOWERS OF HANOI   DISKS: " #DISKS @
 3    . CR CR ;
 4 : ?NUM   ." Number of disks? " QUERY 32 WORD NUMBER ;
 5
 6 : TOWERS   CR ?NUM #DISKS ! STARTED FALSE .HEADER FORPS CR CR ;
 7
 8
 9
10
11
12
13
14
15
```

```
1085 LIST
  0 Towers of Hanoi:  the rules
  1
  2 MOVE-A-DISK recognizes a goal to move a single disk from one
  3 peg to another.  It removes this goal and prints out the
  4 desired move.
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

1086 LIST
  0 Towers of Hanoi: high level word
  1
  2    TOWERS prompts the user for the number of disks and then
  3 procedes to solve the problem.
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```