TR-I-0343

The FLAIL Expert System Shell Manual

John K. Myers

March 12, 1993

Abstract

This manual presents users' documentation for FLAIL, the Fact/rule Language for ATR's Interpreting Telephony Research Laboratories. FLAIL is an inference engine (or, "expert systems shell") program that allows users to write facts and rules, and have the system draw inferences to solve problems.

The FLAIL Expert System Shell Manual

John K. Myers
February 23, 1993
ATR Interpreting Telephony Research Laboratories
Sanpeidani Inuidani Seika-cho Soraku-gun
Kyoto 619-02 Japan
Netmail: myers%atr-la.atr.junet@uunet.uu.net

Abstract

This manual presents users' documentation for FLAIL, the Fact/rule Language for ATR's Interpreting Telephony Research Laboratories. FLAIL is an inference engine (or, "expert system shell") program that allows users to write facts and rules, and have the system draw inferences to solve problems. It was developed to enable ATR to have modifiable LISP source code to an inference engine (which was required for interfacing an inference engine to a LISP ATMS in order to perform plan recognition). Like all inference shells, FLAIL is a general-purpose system that can work with any kind of problem that can be represented using rules and facts; the user is not limited to plan-recognition applications. The current version of FLAIL directly supports forward chaining; backward chaining can be implemented on top of this. FLAIL supports constant facts, rules with variables that can be used in the rules' consequents, hierarchically nested lists for facts, "rest-of" indefinite-count variables, retraction of facts, and escapes to the full LISP operating system.

This manual describes the FLAIL system by itself. For the use of the ATMS system, or the integrated FLAIL+ATMS system, please see the separate appropriate manuals.

[0]Copyright (c) 1989,1993 ATR Interpreting Telephony Research Laboratories.

Contents

1 **The System** 2

2 **Introduction: Programmer's Description** 2

3 **What does an Inference Engine do?** 3

4 **Conceptual Use of FLAIL** 3

5 **Entities: Types of Data Structures** 4
- 5.1 Facts 4
  - 5.1.1 Theory 4
  - 5.1.2 Method 5
- 5.2 Rules 6
- 5.3 Patterns 7
  - 5.3.1 Variables 7
  - 5.3.2 "Rest-of" Variables 7
- 5.4 Lisp Escape Execution Commands 8
- 5.5 Retraction 9
- 5.6 Extras 10

6 **Execution** 10
- 6.1 The Execution Cycle 10
- 6.2 Parallel Execution and Rule Priority Numbers 11
- 6.3 Use of Priorities 15

7 **Examples** 17
- 7.1 Example 1: Sequential Stepping 17

8 **Commands** 19
- 8.1 User Commands 19
- 8.2 User Option Flags 19
- 8.3 Debugging Flags 20
- 8.4 System Configuration Flags 20
- 8.5 Output Stream Variables 20

9 **Command Dictionary** 21

1 The System

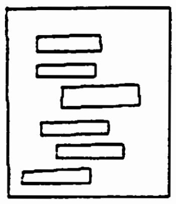
Source and binary for the system is found on the Symbolics Lisp Machines, under file `LM01:>Myers>golden-flail`, with extensions `.lisp` and `.bin` respectively.

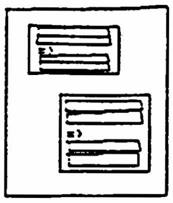
To load the system, type `(load "LM01:>Myers>golden-flail")`.

To run the system, load or type in the facts and rules desired, and then type `(flail)`. See Section 4, Conceptual Use of FLAIL.

2 Introduction: Programmer's Description

FLAIL is a rule-based inference engine that matches facts in a fact data-base against similar patterns in rules in a rule data-base. A rule may have many patterns as its antecedent; if the conjunction of these patterns match in a consistent manner, the rule's consequent is executed sequentially. A consequent is a list of facts, fact retractions, and LISP escape commands. Facts are asserted; retractions are deleted from the fact data-base; and escape commands, which can be any LISP command outside of the FLAIL system, are executed. Patterns may have variables, that are bound when first matched inside the rule but must match the bound value inside that rule thereafter. Bound variables can be used in the consequent, including inside the LISP escape commands. Variables can match atoms or lists; a special type of "rest-of" variable can match sequences of atoms at the end of a list. Facts are unique. The system directly supports only forward-chaining in Version 1.1 (naturally, backward-chaining can easily be implemented on top of this). Rules are fired once for each pattern match. Facts and rules can be asserted in any arbitrary, mixed order, there is no need to forward-reference or predefine items. A fact or a pattern may be an atom, a list, or a nested list. Atomic facts allow the definition of packages of rules. Although the execution order of rule-based systems is typically not guaranteed, rule priorities in FLAIL allow matching rules with high priority to be executed before rules with lower priority. Together, the FLAIL system is intended to provide a clean but powerful package that performs rule-based inferencing, allows nonmonotonic retraction and interfaces with arbitrary outside systems.


The Fact Data-Base


The Rule Data-Base

3 What does an Inference Engine do?

An inference engine is designed to allow relatively simple programming of problem-solving systems by supporting a rule-following paradigm. Inference engines, instead of having a series of subroutines like normal programming languages, have a data-base of *rules*. These rules are of an "if X X then Y Y" form. A rule is an instruction that says if certain conditions (X X) hold, then execute or assert the results (Y Y). The test conditions of the rule are called the rule's *antecedent*; the results are termed the rule's *consequent*. When the rule's antecedents are all valid and the consequents are performed, this is known as *firing the rule*.

The rules work on a data-base of *facts*. Facts are passive data that the system operates on. A rule tests whether all of a series of facts are present in the fact data-base; if so, then the rule typically puts more, different facts into the fact data-base.

Because historically this kind of system was built to deductively infer facts about a situation, this kind of system is called an *inference engine*. Sometimes it is called a *rule-based system*. Because an inference engine is typically the primary system component in so-called "expert systems", this kind of rule-based program, together with support, explanation, and debugging facilities, is sometimes known as an "expert system shell". Note that the inference engine is a system, and therefore cannot do anything by itself; the actual expert system or application program is embodied in the rules and facts that the system works with.

4 Conceptual Use of FLAIL

Inference engines as a whole are very simple and intuitive to use. (The actual problems come in designing the rules so that they do something useful, not in using the system itself.) There are basically only four main commands for the entire system. First, you `reset` the system, clearing out any old facts or rules that might have been left over in the fact data-base or the rule data-base. This is done with the command `(reset)`. Next, you assert facts that you want the system to know about, using the command `(facts ...)` described below. After that, you assert the rules that the system is going to use, by means of the `(rules ...)` command which is also described below. You can assert as many rules and as many facts as you want, and in any order. Each time you call `(facts ...)` or `(rules ...)` the facts or rules get inserted into the respective data-base, along with all the facts or rules that were there previously. (There is no need to predeclare rules, or to have to declare facts before rules, as there is with some systems.) After you have finished loading all the facts and rules, you run the system, by using the command `(flail)`. The system keeps executing rules until none of them fire any more.

5 Entities: Types of Data Structures

We now go into a discussion of the actual entities (conceptual data structures) that are used by FLAIL. These consist of the facts, the rules, the patterns used by the rules to match the facts, and the two kinds of variables inside the patterns.

5.1 Facts

5.1.1 Theory

Facts are the basic data of the system. A fact can be an atom, a list of atoms, or even a hierarchical list of atoms and lists. Examples of facts (with explanatory notes) include the following:

`conversation-package` Single LISP atoms are allowed.

`(guests-turn)` Lists with one atom are also allowed.

`(The guest wants to talk)` Lists with multiple atoms are the most commonly used,

`(The (guest named Kazuko) said (Where is the conference?))` however it is also possible to have nested lists in a single fact.

`(I want (You to tell me (What is (your name))) please)` Lists can be hierarchically nested as deeply as is necessary.

Remember that in LISP, the hyphen "-" is treated as a letter and can appear in the middle of a single name just like any other letter.

Of course, a fact, just like anything else inside a computer, has no intrinsic meaning. The only meaning associated with a fact is what the rest of the computer program can do with it. For instance, a person can enter the fact (Takeshita is President of the USA) and the computer will accept this happily. We use the simple name "fact" as a convenience to refer to the assertions that the system works with; whether the fact is actually true in the real world or not does not matter to the computer. In fact, facts in FLAIL are not true or false, and are not interrelated; a fact is either IN THE SYSTEM, in which case it is KNOWN, or it is simply not in the system (unknown), perhaps because the user has not typed it in yet or it has been retracted.

Since facts are not interrelated, there is nothing to stop the user from also entering (Bush is President of the USA) into the fact data-base, (unless there is some special rule that the user has created that detects multiple presidents and does something about it). However, facts are unique; if the user again enters (Takeshita is President of the USA) as a fact, there are not two facts starting with (Takeshita...) in the fact data-base that are exactly the same; there is only one–the second one overwrites the first one, if it is *exactly* the same fact.

Facts are treated as constants. Although it is lexically possible, facts should not have any variables (represented by symbols beginning with question marks, which of course includes a single question-mark by itself) in them, because they have no semantic meaning. That is, if you write a fact with a variable in it, e.g.

```
WRONG:  (The guest said ?something to ?someone)
```

you are quite probably making a mistake. The system will allow you to assert such a fact, but it will complain to you (unless the system variable *watch-variables-in-facts* is set to NIL. Don't do this.)

5.1.2 Method

Facts are asserted using the (facts ...) command. This takes a series of facts as its argument. Although you can call facts from inside a program, for example:

```
(setq my-fact '(This is a fact))
```

```
...
(facts my-fact)
```

it is much more common to call this command from the top level, with a list of literal arguments. Since `facts` evaluates its arguments, in this case it is necessary to put a quote before each fact:

```
(facts
'conversation-package
'(guests-turn)
'(The guest wants to talk)
'(The (guest named Kazuko) said (Where is the conference?))
'(I want (You to tell me (What is (your name))) please)
)
```

Of course, we can call `facts` again and add more facts to the ones that we just asserted. We can add more facts to the system at any time, even after we have added rules or after we have run the system. And, naturally `facts` can take a single fact as an argument:

```
(facts '(Another additional fact))
```

Be careful that you do not put a set of parentheses around all of the fact arguments–`facts` takes a *series* of arguments, not a *list*. If you type

```
WRONG:  (facts '( '(fact1) '(fact2) ))
```

you will get a single fact asserted which is a list of two items (each consisting of a quoted list), which is probably not what you wanted.

Since facts are treated as constants composed of atoms and hierarchical lists, they can store just about anything. It is particularly interesting to store facts that take the form of rules; in this case, one can implement a meta-system, using the rules in the rule data-base, that performs backward chaining, heuristic search, etc.

5.2 Rules

Rules are the basic "program instructions" of the system. A rule consists of a situation to recognize, which is a list of facts called the *antecedent*, plus a list of actions to take, called the *consequent.* The rules operate on facts; if all of the facts in the antecedent are present in the fact data-base, that rule is *fired* (executed) and all of the actions in the consequent are performed. Actions usually consist of adding more facts to the fact data-base, although actions can be retracting (deleting) a fact from the fact data-base, or executing an arbitrary Lisp function.

5.3 Patterns

The way that a rule matches and asserts facts is through the use of patterns. There are two kinds of patterns, antecedent patterns and consequent patterns. Antecedent patterns look like facts; they are input patterns to the rule, used for matching. Consequent patterns also look like facts, but they are output patterns for asserting facts into the fact-base or for executing functions.

5.3.1 Variables

Ordinary FLAIL variables are distinguished by atom names that start with the "?" character. They can match atoms or entire sublists in a fact.

Examples of variables inside patterns, and legal matches, includes:

(?a) matches a single item in a list, such as (my-fact) or ((single sublist in list)). However, it will *not* match (two facts) or ((two)(sublists)).

(my ?what) matches (my statement) or (my (anything here)), but not (your statement).

?single-var matches fact `atomic-fact`.

(?what ?what) matches (a a) or ((foo bar)(foo bar)), but not (a b).

Variables are bound to their matches inside a particular rule; all of the matches to a single variable must be consistent.

5.3.2 "Rest-of" Variables

In addition to the regular variables, there are special, "rest-of" variables that start with a "+" character. These variables match one or more atoms or sublists in a fact, up to the end of a list; they must occur as the last member in a list or sublist inside a pattern.

Examples of this kind of variable in a pattern, and legal matches, includes:

(+a) ...matches fact... (x y z)

(a +b) ...matches fact... (a x y z)

(I said (you +what) right) ...matches fact... (I said (you x y z) right)

(+a) ...matches fact... (sequence (with (sub)) lists)

In all of these cases but the last one, the "rest-of" variable is bound to the sequence x y z. Note that this is *not* the *list* (x y z); if the variable is used in the consequent, the sequence is spliced in where the variable was.

Example: In a consequent pattern,

(you said +what) ...expands to (you said x y z), NOT to (you said (x y z)).

For this reason, +variables in consequents must be inside a list or a sublist. However, unlike antecedents, they do not have to be the last member of the sublist.

Examples of incorrect usages of +variables in antecedents include:

wrong: +a +Variable not inside parenthises.

wrong: (and +now what) +Variable not last item in sublist.

5.4 Lisp Escape Execution Commands

Besides facts that get asserted, it is possible for a rule to have arbitrary Lisp commands in its consequent that get executed when the rule gets fired. Because these are commands that are not part of the closed FLAIL system itself, they are called "escape" commands. Escape commands can execute any legal Lisp function, macro, or special form. The assertions and the escape commands are performed in the order in which they appear in the consequent.

Escape commands are indicated by the first atom in the consequent pattern list being a "`!`" character. Thereafter, the rest of the command, as regularly typed in to a Lisp Listener, is presented. However, like fact assertions, the escape command can contain FLAIL variables. These are macro-instantiated to literals before the command is executed. Thus, the escape commands can use the bindings of FLAIL variables.

Examples of Lisp escape execution commands, to appear in a rule's consequent, include:

(! format T "My variable is A." (quote ?what)) Prints binding of FLAIL variable `?what` inside rule.

(! setq my-var (quote ?what)) Sets Lisp global variable `my-var` to the binding of FLAIL variable `?what`.

5.5 Retraction

In addition to asserting facts and executing Lisp escape commands, it is also possible to retract facts–that is, erase them from the fact database. Retraction is done by using the `!retract` command pattern in the consequent of a rule. A fact to be retracted is presented just as it normally is, inside the !retract pattern.

Examples:

(!retract (an-atom-in-a-list)) ...retracts fact: (an-atom-in-a-list)

(!retract atomic-fact) ...retracts fact: atomic-fact

(!retract (I said ?what) ...uses binding of variable `?what` and then retracts the instantiation of fact (I said ?what).

Retracting a fact also immediately pulls all rule instantiations depending on that particular fact out of the execution stack (see Section 6.1).

For convenience, `!retract` is also implemented as a top-level function, that the user can call from Lisp. However, in this case its argument is evaluated, and must be quoted.

5.6 Extras

Rules can also contain an optional documentation string. This is usually listed before the antecedent patterns, and is used purely for documentation, debugging and informational purposes. It is a good idea to put a textual description of what the rule is expected to do in this place, because sometimes the intention behind a rule can be hard to understand, especially if the rule has a bug.

In addition, if the first item in the rule is an atom (not a list or a string), it is treated as the variable-name of that rule. A global variable is created with that name, and the rule is stored in that variable (besides being stored in the system rule data-base, as usual). Naturally, the system does not include the variable-name as part of the antecedent patterns of that rule. Watch out for the mistake of creating a fast rule without a variable-name or a documentation string, and then changing it to start with a package-name (single-atom) pattern–this will be treated as the rule's variable-name.

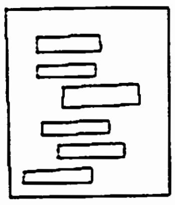
Rules can also have optional priorities. A priority is a single number, usually placed after the inference arrow ("=>"). Be careful not to put the number directly after the arrow, without an intervening space–the parser will think that the arrow plus the number is one symbol. A priority should be an integer, but it can be negative. The use of priorities is discussed in depth in Section 6.2. Rules without explicit priority numbers get the implicit priority of zero (0).

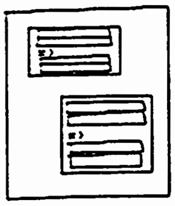
6 Execution

The previous section has discussed the kinds of data that the system works with. This section will now discuss how FLAIL uses this data to run.

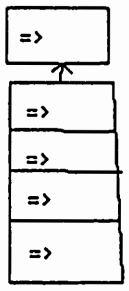
6.1 The Execution Cycle

The FLAIL system uses an execution stack of rules. The top rule is examined; if all of the antecedents match consistently, the rule is fired,


The Fact Data-Base


The Rule Data-Base

and the consequents of the rule are asserted or executed in order. If the antecedents do not match, this examination of this rule is discarded. After the system is finished with the current rule being examined, the stack is popped, and the next rule is examined.



The Execution Stack and the Rule Being Examined

The rules that the stack stores are not actual rules, but particular instantiations consisting of a pointer to a rule in the data-base plus a (possibly null) list of rule variable-bindings. Thus, when the stack discards a rule after it is finished firing, the actual rule is still in the rule-base; only that particular instantiation is discarded. A single rule from the rule-base may have many instantiations on the stack at the same time, corresponding to different variable bindings.

Individual rule patterns are matched when a new fact is asserted into the fact data-base. Of course, a rule has a series of individual patterns in its antecedent. If a new fact triggers a rule, such that each of the patterns in the rule's antecedent matches some fact, then that instantiation of the rule is queued onto the stack. The system computes *all possible permutations* of matching facts for that rule that include the new fact, and queues each of these on the stack as a separate instantiation. Currently, when a rule instantiation comes to the top of the stack and is examined, although each of the rule patterns matches, it is not clear that they all match together–i.e., that the variable bindings amongst the various patterns in the antecedent are consistent. This testing must be done somewhere, and is currently done at the top of the stack by examining the rule, as mentioned previously.

The stack is actually implemented as a heap, i.e. an ordered list with priorities. The priority of a rule is used to order it in the heap. Thus, rules with high priorities are all fired before rules with lower priorities. Rule priorities and their use will be discussed in further detail in the next section.

6.2 Parallel Execution and Rule Priority Numbers

Rule-based systems are significantly different from conventional programming languages. The execution of the rules (which correspond

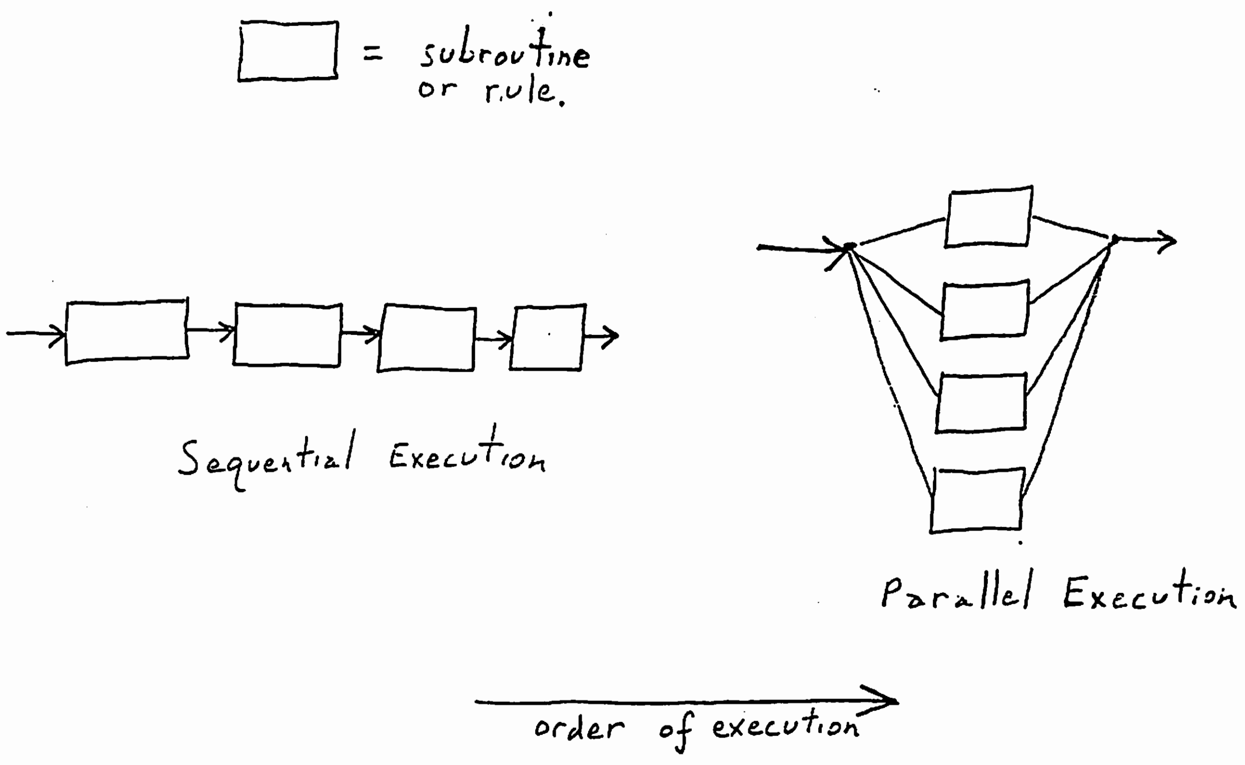


Figure 1: Customary Language Sequential Execution vs. Rule-Based System Parallel Execution.

conceptually to subroutines) is designed to proceed in (virtual) parallel order, instead of in the sequential order found in customary programming languages. See Figure 1. In particular, *the order in which any two specific rules are executed is not guaranteed. In fact, the order in which the rules in the same program are executed is not guaranteed to not vary from one execution run to the next.*

This is one of the main powers of rule-based systems. The rules that are important are used; the ones that are not important are ignored. New rules can be added at the end of previous rules, without having to sort them into a program structure. The system decides which rules are applicable, and performs inferences with those rules only.

This presents no problems whenever the system acts in a monotonic fashion, and whenever the execution of rules' consequents have no side-effects. As long as the actions of the system consist only of adding facts to the data-base, `!retract` is not used, and no LISP escape commands to user functions are called, it does not matter what order the rules are executed in–all the rules that are important will eventually get executed anyway. This makes programming rule-bases very convenient.

However, such is not typically the case. Generally, anywhere where there are printouts, anywhere where the system retracts something incorrect or saves search time by retracting an unproductive search branch, anywhere where FLAIL calls a user system that does something, or any time that temporal order is important, the unspecified ordering of rule-based systems is disadvantageous.

For instance, imagine a conversation between two people. The system is initialized with one fact for each utterance in the conversation, corresponding to the literal content of the utterance. We enter a single rule:

```
(rules '("Rule for printing out conversations."
             ?utterance
          =>
            (! print ' ?utterance)
        )
)
```

that is designed to print the whole converation out. This will work; this rule will get fired once for every utterance in the fact data-base, printing it out. However, because the order of execution of rules is not guaranteed, the utterances in the conversation will be processed and printed out in random order. The last utterance could be printed out first, last, or in the middle. Since much of the contextual information found in a conversation is derived from the order of the utterances, obviously this is an intolerable situation.

What is needed is a method of partially defeating the non-guaranteed virtual-parallel order of execution of the rules. This is supplied by the optional *rule priority number*. Rules are executed in the order of their priorities, highest to lowest; all valid rules with a high priority are guaranteed to be executed before other rules that have lower priority. If many rules with low priority are executing, but a rule having a high priority all of a sudden becomes valid (because its antecedents have suddenly become true and consistent), that rule is immediately fired.

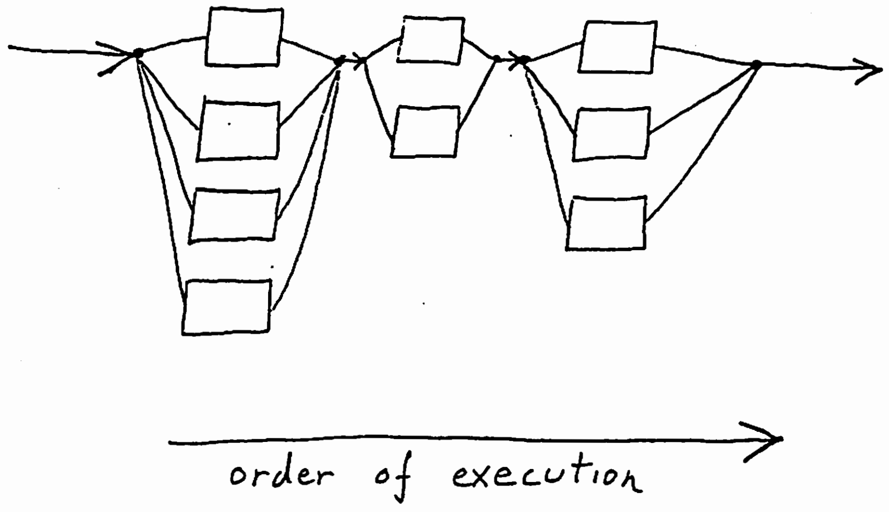


Figure 2: Semi-Parallel Execution Using Rule Priorities.

Rules without a priority number are assigned an implicit priority of zero.

Although rule priority numbers can be any kind of rational number, there is no reason to make them anything other than integers. High numbers (e.g., "10") will execute before most other rules; low numbers (e.g., "-8") will execute only after all of the other rules have died out. The scale of rule priority numbers is arbitrary, and one may use any priority numbers desired; the only thing that matters is their relative order.

A rule-based system with priorities executes its rules in a semi-parallel fashion. See Figure 2. Notice that within the same priority, execution is still in parallel and the order is still non-guaranteed. However, all the rules at a particular priority must finish firing before the rules at the next priority down are allowed to start. This allows the system to retain the advantages of non-specified ordering and use-when-needed rules, while permitting specification of order in the special cases where it is necessary.

6.3 Use of Priorities

The question arises: when should priority numbers be used? The answer is that priority numbers should not be used unless it is obvious that they are absolutely needed. This only occurs with temporal ordering problems and deadlocks, where one section of the rule program might be trying to delete a piece of data before another section is done with it. In general, this happens with system control-flow functions, and not with user applications. Examples are discussed below.

How does one choose priority numbers? The absolute, actual value of a priority number has no meaning; it is only the relative value, when compared against other rules, that matters. Priority numbers should be assigned to rules based upon the mandatory temporal ordering of the rules. "Must run before" means *has to have a higher priority*; "must run after" means *has to have a lower priority*. Examples of this type of situation, where priority numbers must be assigned, include:

- Sequential handling. When items must be handled sequentially, it is necessary to implement special routines that assert the next item in the sequence after all the processing on the current item is finished.
- Preprocessing data. Parts of the system must delete or modify incoming data before the rest of the system can be allowed to work on it.
- Postprocessing data. Parts of the system must wait until the rest of the system is completely finished, before they can be allowed to work on the data as a whole.
- Traps and interrupts. If something is extremely important, it must interrupt the normal flow of control of the system and run before all the rules that the system is currently executing. Note that returning from an interrupt is trivial, as once the system finishes processing the rules in the interrupt it automatically continues processing the previous rules that are still valid.
- Flow of control: Switching "packages" (subgroupings of rules). All of the rules inside a particular conceptual grouping, or *package*,

must be completely finished before other packages are allowed to run. The control rules that switch to a new package must run after all the current package(s)'s rules are finished.

- Deleting inconsistent facts. If a fact is found to be inconsistent, it must be deleted before other rules operate on it and propagate it further. The rules that detect and delete inconsistent facts should therefore run at a higher priority than the other rules.
- Trimming branches on a search tree. Branches that are dead and must not be explored further should be trimmed before they are expanded.
- Presenting the answer from computed results to a problem. The rules that assemble and print out an answer are a special case of postprocessing.
- Flag-setting. If an important flag must be set that determines the manner in which the rest of the system processes the data, this flag should be set first.
- Default reasoning/exception handling. Priorities can easily handle exception handling, or marked items that block defaults, by handling the exceptions first and then removing the item or an "unhandled" flag. Defaults are then handled afterwards, if the item still needs to be processed.

Note that in each of these cases, there is a clear *system* need for temporal processing order. The capability to perform the desired behavior could not be constructed in a system that operates purely in parallel.

It is important to distinguish these mandatory system needs for priorities from *user application* desires for priorities. In most cases, the actual rules that make up the user's application itself should run all at the same priority. Although it is certainly possible to specify different priorities for different segments of the user's application, in general it is a difficult problem to assign the priorities properly. The user is therefore on his or her own in this regard.

7 Examples

7.1 Example 1: Sequential Stepping

```
(setq *watch-facts* T) ;Watch what's happening.

(rules ;Start entering two rules.
;These are the Step rules.
;They allow things to be sequentially asserted.

;Each rule must be quoted.
'(step-rule1  ;Rule-variable--put this rule in this var.
  "Sequentially asserts facts." ;Documentation string for this rule.
      (step ?x +y) ;If you see a "step" fact,
      =>    ; then
      (!retract (step ?x +y)) ;retract it, and
      ?x ;assert the first part of it, and
      (step +y) ;assert a new, smaller "step" fact.
      )
;Actually, normally this would be run
;at a low priority...

;We need one more rule to tie off
;the recursion.

'(step-rule2 ;Store rule in var step-rule2.
  "Ties off the last fact in the sequence." ;Documentation string.
   (step ?x) ;If you see a "step" with only one arg,
   =>  ;  then
   (!retract (step ?x)) ;retract it, and
   ?x ;assert its argument.
   )
) ;End of "rules".

;Now, let's test it out.
(facts ;We'll enter one fact.
;Each fact must be quoted, too.
   '(step ;This is a "step" fact:
(First Fact)
(Second Fact)
(Third Fact)
    ) ;End of "step"
) ;End of "facts".
(STEP (FIRST FACT)(SECOND FACT)(THIRD FACT))  ;System comes back with
;a report of the one fact entered,
NIL ;and "facts" returns NIL.

(flail) ;Fire the system up!
Retracting fact: (STEP (FIRST FACT) (SECOND FACT) (THIRD FACT)).
-> (FIRST FACT) ;First fact is asserted.
-> (STEP (SECOND FACT) (THIRD FACT))  ;Step is reiterated.
Retracting fact: (STEP (SECOND FACT) (THIRD FACT)).
-> (SECOND FACT) ;Second fact is asserted.
-> (STEP (THIRD FACT))
Retracting fact: (STEP (THIRD FACT)).
-> (THIRD FACT) ;Third fact is asserted.
NIL ;Flail runs out of rules and returns.

;Now, we want to see the results.
(print-facts) ;Print all the facts in the fact-base.
FACTS: ;Note that they are all there.
(SECOND FACT) ;But, they're not in order!
(THIRD FACT)
(INITIALIZE) ;This one is put in by (reset).
(FIRST FACT)
NIL ;print-facts returns.

;End of demo.
```

8 Commands

8.1 User Commands

(reset) Clears the system out.

(facts *'fact1'fact2* ...) Enters a series of facts separately into the data-base. Since the facts are evaluated, each fact must be quoted if you are typing the command in directly.

fact description: Facts can consist of:

- a-single-atom
- (an-atom-in-a-list)
- (a list of atoms (possibly with sublists))
- ((lists can (be (nested))(((to an arbitrary depth)))))

(rules *'rule1 'rule2...*) Enters a series of rules separately into the rule data-base. Since the rules are evaluated, each rule must be quoted if you are typing the command in directly.

(!retract *fact*) This function retracts a fact from the fact data-base. It can be used by itself, or appear in the consequent list of a rule.

(! *Lisp-escape-function arg1 arg2 ...*) This pattern, when entered in the consequent of a rule, allows the rule to temporarily escape from FLAIL and execute an arbitrary Lisp function. Rule consequent patterns are asserted or executed in the order in which they appear in the rule.

8.2 User Option Flags

watch-facts This flag makes the system print out each new fact that gets asserted.

flail-stack-count This flag tells FLAIL to simply print out an integer describing the length of the stack every time a rule is examined. It is useful for telling how deep the stack is getting, and what percentage of the rules are actually firing.

8.3 Debugging Flags

watch-execution-dots This flag makes the system print out a dot for every rule that is examined but fails to fire, and a star for every rule that fires successfully.

flail-stack-debug This flag tells FLAIL to dump the stack every time a rule is examined.

8.4 System Configuration Flags

use-nice-assertion Flag governing whether checks are performed on facts to be asserted, or whether facts just get put in straight without checking.

watch-redundant-facts This flag makes the system complain if you put the same fact into the data-base twice.

watch-variables-in-facts This flag makes the system complain if one of the atoms in an asserted fact looks like a variable, i.e. starts with a question-mark. It should always be T.

8.5 Output Stream Variables

OS This variable holds the flail system standard output stream. Normally it is set to T, to print out on the screen; however, it may be set to a user-allocated stream, to print to a file.

DS Flail system debugging output stream.

ES Flail system error output stream.

9 Command Dictionary

(! *Lisp-escape-function arg1 arg2 ...*) This pattern, when entered in the consequent of a rule, allows the rule to temporarily escape from FLAIL and execute an arbitrary Lisp function. Rule consequent patterns are asserted or executed in the order in which they appear in the rule.

DS Flail system debugging output stream.

ES Flail system error output stream.

fact description: Facts can consist of:

- a-single-atom
- (an-atom-in-a-list)
- (a list of atoms (possibly with sublists))
- ((lists can (be (nested))(((to an arbitrary depth)))))

(facts *'fact1'fact2 ...*) Enters a series of facts separately into the data-base. Since the facts are evaluated, each fact must be quoted if you are typing the command in directly.

flail-stack-count This flag tells FLAIL to simply print out an integer describing the length of the stack every time a rule is examined. It is useful for telling how deep the stack is getting, and what percentage of the rules are actually firing.

flail-stack-debug This flag tells FLAIL to dump the stack every time a rule is examined.

OS This variable holds the flail system standard output stream. Normally it is set to T, to print out on the screen; however, it may be set to a user-allocated stream, to print to a file.

(reset) Clears the system out.

(!retract *fact*) This function retracts a fact from the fact data-base. It can be used by itself, or appear in the consequent list of a rule.

(rules *'rule1 'rule2...*) Enters a series of rules separately into the rule data-base. Since the rules are evaluated, each rule must be quoted if you are typing the command in directly.

use-nice-assertion Flag governing whether checks are performed on facts to be asserted.

watch-execution-dots This flag makes the system print out a dot for every rule that is examined but fails to fire, and a star for every rule that fires successfully.

watch-facts This flag makes the system print out each new fact that gets asserted.

watch-redundant-facts This flag makes the system complain if you put the same fact into the data-base twice.

watch-variables-in-facts This flag makes the system complain if one of the atoms in an asserted fact looks like a variable, i.e. starts with a question-mark. It should always be T.

References

[WN88] John R. Walters and Norman R. Nielsen. *Crafting Knowledge-Based Systems: Expert Systems Made (Easy) Realistic.* John Wiley & Sons, New York, NY, 1988.

Index

"+" variables 7
"?" variables 7
antecedent 3
antecedent 6
consequent 3
consequent 7
facts, method of use 5
facts, theory of use 4
facts 3
firing rules, definition of 3
(flail) 4
high priority 15
inference engine 3
low priority 15
number, rule priority 13
numbers, priority, use of 15
package 15
priority number, for a rule 13
priority numbers, use of 15
priority, use of high and low 15
reset 3
rule priority number 13
rule-based system 3
use of priority numbers 15
variables 7