Representing a rule-based expert system in Lisp: Part 2, forward chaining

Christopher Buckingham

If you have been struggling to complete the first part of this tutorial, check the help on BlackBoard for this lab, which explains how to go about the tasks. It provides a suggested rule base and a series of function calls that define the building blocks of the system. These calls are repeated below for convenience but note that the facts are NOT realistic (as far as I know, there are no ungulates with feathers): they are simply used to test whether the functions are working. The idea is to load in some test variables (i.e. *facts and some test rules) so that it is easy to pass them in to the functions for checking their operation. The following calls in Lisp show you what I mean and how the facts and test rules are used to check the functions you have written:

```
   ------------------ OUTPUT TEST FILE -------------------------
[22]> *facts
((FEATHERS Y) (LAY-EGGS Y) (FLY N) (UNGULATE Y)(BLACK-STRIPES Y))

[13]> test-rule1
(BIRD ((FEATHERS Y) (LAY-EGGS Y)))

[14]> (get-conditions test-rule1)
((FEATHERS Y) (LAY-EGGS Y))

[15]> (get-conclusion test-rule1)
BIRD

[16]> (condition-known '(feathers y))
(FEATHERS Y)

[17]> (condition-true '(feathers y))
T

[17]> (condition-true '(feathers n))
NIL

[21]> (triggered-rule test-rule1)
(BIRD ((FEATHERS Y) (LAY-EGGS Y)))

[22]> (get-triggered-rules *rules)
((ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))) (BIRD ((FEATHERS Y) (LAY-EGGS Y))))

[20]> *facts
((FEATHERS Y) (LAY-EGGS Y) (FLY N) (UNGULATE Y) (BLACK-STRIPES Y))

[21]> (fire-rules (get-triggered-rules *rules) *goals)
(ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y)))

[22]> *facts
((ZEBRA Y) (FEATHERS Y) (LAY-EGGS Y) (FLY N) (UNGULATE Y)(BLACK-STRIPES Y))
```

This tutorial builds on the functions above to produce the reasoning process and user interface. It is up to you how you do it but the following section gives function names and parameters, with accompanying comments, that are one possible avenue of exploration. You need to write the body of the functions so that they behave as specified in the comments.

1 Rule names and parameters with comments

Note that we will treat *facts as a shared "message board" or "blackboard", in the parlance of AI, where any function can access and update it without having to pass it as a parameter. This simplifies the function calls even though it is usually a bad idea to change variables that are not passed in as parameters. An exception is made in this case because of the clearly understood semantics of the expert-system architecture. Problems are avoided by stating the convention clearly and using globals that are easy to find in the code. A comment for any function changing the global should always be used as well.

```
;; Adds rule conclusions to the global fact base (working memory).
;;
;; Returns the rule that generates a goal or nil.
;;
(defun fire-rules (triggered-rules goals)
  <function body for you to define>

;; Keeps firing triggered rules and updating the fact base until there
;; are no more untriggered rules or the goal has been found.
;;
;; Returns the goal rule or nil.
;;
(defun forward-chain (triggered-rules rules goals)
  <function body for you to define>

;; main program starts by forward chaining but if the conclusion has
;; not been found, it reverts to asking the user for more facts (to be
;; done in a later tutorial).
;;
(defun run (rules goals)
  <function body for you to define>

;;SOME USEFUL FUNCTIONS FOR OUTPUTTING AN ANSWER

;; Returns a list of the properties of a rule's conditions that have a
;; 'y' answer
;;
(defun get-yes-conditions (rule)
  (let ((conditions nil))
    (dolist (condition (get-conditions rule) conditions)
      <code for you to define>

;; Returns a list of the properties of a rule's conditions that have a
;; 'n' answer
;;
(defun get-no-conditions (rule)
  (let ((conditions nil))
    (dolist (condition (get-conditions rule) conditions)
      <code for you to define>
```

Note the repetitive nature or the get-yes-conditions and get-no-conditions functions. When you see code repeated like this, you know that it is inefficient. What you should do is merge them into one function and pass a parameter that tells the function whether it should return yes or no conditions:

```
(defun get-conditions-with-same-answer (answer rule)
  (let ((conditions nil))
    (dolist (condition (get-conditions rule) conditions)
      <code for you to define>
```

Then it returns the conditions if their value (y or n) matches the value of the "answer" parameter.

2 The main program

The steps for running the system are very much open to choice but here is a suggestion for one that works on a given starting set of facts but does not try to find out any additional facts from the user. Basically, you need to define the main program function, `run`, that operates on the starting facts that are loaded into clisp as a global variable. The function uses the rule base and the goals to work out what the animal is from the known facts:

```
(defun run (rules goals)
  <function body for you to define>
```

The following steps are suggestions for how you can design and implement the main program. In other words, these are the internal functions of `run`.

1. Tell the user what facts are known at the start by the system:

   ```
   (format t "~%The current facts before applying rule-based reasoning are:
   ~a~%" *facts)
   ```

   (a) This uses the `format` function that has the first argument saying where the output of the string will go, a string as the second argument with some control characters indicating variable values, and the variables matching the control characters. More information will be given on this later but for now, the above function call can be read as follows:

   ```
   t means output the string to the termianl.
   ~% means output a line break
   ~a means put a variable value in here, with no additional formatting
   *facts is the variable that is evaluated and put in place of the ~a.
   ```

   (b) You can have several variable control characters in the string where each one is matched in order with the same number of variables after the string.

2. Execute the `forward-chain` function and pass the value returned to a local variable.

3. If forward chaining has found a goal rule

   (a) Get the yes conditions;

   (b) Get the no conditions;

   (c) Tell the user what the new facts are after firing the goal rule;

   (d) Tell the user what animal has been identied, using the conclusion of the goal rule;

   (e) Tell the user you know this because the animal has the known conditions of that goal rule.

4. If forward chaining has not found a goal rule, tell the user you can't identify the animal.

An example of some output is the following

```
[5]> (run *rules *goals)

The current facts before applying rule-based reasoning are:
((GIVE-MILK Y) (HAIR Y) (CHEW-CUD Y) (BLACK-STRIPES Y))

The new facts after rule-based reasoning are:
((ZEBRA Y) (UNGULATE Y) (MAMMAL Y) (GIVE-MILK Y) (HAIR Y) (CHEW-CUD Y)
(BLACK-STRIPES Y))

and your animal is a ZEBRA because it has the following conditions:
(BLACK-STRIPES UNGULATE).
```

Next week, we will look at how backward chaining can provide a better explanation for why the system thinks the animal is a zebra.