# Representing a rule-based expert system in Lisp

Christopher Buckingham

These tutorials will create a forward and backward chaining expert system in lisp, starting with a forward chaining one. Each rule in Section 1 is a separate rule in your rule base. So if more than one rule has the same conclusion then they are considered as disjunctions:

```
IF these conditions OR these conditions

THEN this conclusion.
```

which is implemented as:

```
IF these conditions
THEN this conclusion

IF these conditions
THEN this conclusion
```

## 1 rules

- A mammal has hair and gives milk.
- A bird has feathers and lays eggs.
- A carnivore is a mammal that eats meat, has pointed teeth and forward-looking eyes.
- A carnivore is a mammal that eats meat and has claws.
- An ungulate is a mammal with hoofs. and which chews the cud.
- An ungulate is a mammal that chews the cud.
- A cheetah is a carniverous mammal with a tawney colour and dark spots.
- A tiger is a carniverous mammal with a tawney colour and black stripes.
- A giraffe is an ungulate with a long neck, long legs, and dark spots.
- A zebra is an ungulate with black stripes.
- An ostrich is a bird which does not fly, has a long neck, long legs, and a black and white colour.
- A penguin is a bird which does not fly but can swim and which is black and white.
- An albatross is a bird which is a good flier.

## 2 Convert the rules into Lisp

When considering the design of the rule-base, you need to think about how the rules will be processed. What information will need to be found quickly in the rules? How will the pattern matching take place? For example, backward chaining needs to match rule conditions with rule conclusions; finding the conclusion of a rule in the list is therefore going to be an important and frequent task. Likewise, forward chaining needs to match facts in the working memory with rule conditions so matching facts and conditions will be a frequent task. Your representation of rules in Lisp lists should facilitate these tasks.

One of the most powerful accessors of information in a list that we have used so far is `assoc` that pulls out a property-value pair from an association list, which is a list of these property-value pairs. If we made the set of rules an association list with the conclusion as the property, it would be easy to search the rules for ones with a particular conclusion.

### 2.1 Create a rule

1. Create a new file in Emacs called `expert.lisp` that will hold your Lisp expert system.
2. Convert one of the rules into a property-value list where the first member is the property (the rule conclusion) and the second member is an association list of each condition and value pair such as `(give-milk y)` or `(hair y)`. The first rule, for example, would be:

   ```
   (mammal ((hair y)(give-milk y)))
   ```

3. Set up a test variable that is a temporary rule such as `test-rule` and give it the value of your new rule. This will be used to test some functions that operate on single rules. When you are satisfied that you have the right structure for your rule and that it is working well with the functions operating on it, you can convert all the rules into the same structure. But you need to do this later, after you have created a suitable rule structure that works; otherwise you will have to edit all the rules if a change is needed.

### 2.2 Functions operating on a single rule

For each of the functions below, test them on your `test-rule` global variable that is used only for development of the system. The final system will have it deleted or commented out.

1. Write a function called `get-conclusion` that has one parameter, a rule, and returns the conclusion as the symbol name. For example if the conclusion of your test-rule happened to be a mammal:

   ```
   >(get-conclusion test-rule)
   >MAMMAL
   ```

2. Write a function called `get-conditions` that has one parameter, a rule, and returns an association list of the rule conditions.
3. Write a function called `get-yes-conditions` that has one parameter, a rule, and returns an association list of the rule conditions that have a value of 'y' (i.e. yes).
4. Write a function called `get-no-conditions` that has one parameter, a rule, and returns an association list of the rule conditions that have a value of 'n' (i.e. no).

   These last two rules will help with the user dialogue by enabling the interface to say an animal has got these conditions (the 'yes' ones) but not these conditions (the 'no' ones).

### 2.3 Set up the global rule base

When you are satisfied that your rule structure is correct and you can easily access each element:

1. Convert all the rules in Section 1 into the same lisp structure type and put them into a single list.

2. Set the value of a global variable, `*rules`, to be this list of rules. In other words, you will write in your Emacs file `(setf *rules <list of rules>` where `<list of rules>` is your rule base you have just created: `(rule1 rule2 ... ruleN)`.

3. `*rules` is a global variable because it is not inside a `let` statement so will be loaded into the lisp interpretor and be accessible to any function when you load the file into clisp. We use the * symbol in front of the name simply as a convention to distinguish variables that are global rather than local. Remember that local variables are declared inside a function using the `let` function (see the lisp functions document previously given to you and available on Blackboard). They only have a scope inside the `let` so disappear when the function (strictly, the `let`) is exited. Global variables sit in the lisp interpreter and can be accessed by any functions. They should be used sparingly but are useful for large knowledge bases that are the central "database" of a system, which is the case for the rules knowledge.

### 2.4 Identify the rule-base goals

1. The system needs to know which of the conclusions are goals where it has reached an answer and can stop. So create a second global variable called `*goals` and make it a list of the goal conclusions. These will be the conclusions that identify a specific animal.

## 3 Functions to match and check the truth value of a rule condition

Now that you have set up your global rule base, think about how you need to use it. A rule-based expert system needs to know whether a rule condition is in the fact base and whether the value of the fact matching that condition is the same as the one for the rule. This suggests two functions are needed.

1. Set up a global variable called `*facts` that will be used as a convenient way of testing your rules on the working memory (facts). It will be an association list of properties and values (e.g. `((hair y)(lay-eggs n))` that match rule conditions.

2. Define a function called `conclusion-known` that takes two parameters, a rule and a list of facts, returning the rule conclusion if it is in the list of facts or `nil` otherwise, as follows for an 'Ungulate' rule where the facts have `(ungulate y)` in them:

```
(conclusion-known rule *facts)
>UNGULATE
```

3. Define a function called `condition-known` that takes a rule condition (e.g. `(hair y)`) and a list of facts, returning the condition and its value if it is in the list of facts or `nil` otherwise.

4. The rule condition could be in the facts but have the wrong value and we will need to know this when trying to prove a rule, which requires knowing whether each condition is true. So you need to write a rule called `condition-true` that takes a rule condition and the facts. It returns `T` if the condition is in the facts and its value matches the one in the facts. Use `eql` for checking the equality of two symbols because the value of conditions is either `y` or `n`. This function should use the `condition-known` rule to get the condition from the facts.

Now your system can check whether a rule condition is in the facts and is true. However, a rule is only "triggered" if *all* the conditions are true. You need a `triggered-rule` function to find out.

## 4 Triggered rule functions

1. Define a function called `triggered-rule` that takes a rule and the facts as parameters. It returns the rule if the conditions match the facts AND the rule conclusion is not already known (there is no point triggering the rule if its conclusion is in the fact base). It returns nil or the triggered rule.

2. Define a function called `get-triggered-rules` that takes the list of rules and the facts and returns all the triggered rules.

In our forward-chaining rule-based system, we will only have the conflict-resolution strategy that is built into the triggered-rules function we don't fire triggered rules that already have a conclusion in the facts. We will fire all the other rules.

## 5 Fire rules and add facts to the list of facts

1. Write a function called `add-facts` that takes a property-value fact list and the list of known facts, adds the single property-value fact to the list of facts and returns the new list of facts (i.e. the updated working memory). For example, if your rule conclusions are single symbols (e.g. `ungulate, penguin`), then when a rule is fired, it will add this symbol and the value of yes, or 'y', to the list of facts: `(ungulate y)` for example, will be added to the list of facts.

2. Next week, you will write a function called `fire-rules` that updates the list of known facts. For now, write the English steps that defines the algorithm to be used.