Representing a rule-based expert system in Lisp, Part 4: explaining the reasoning process

Christopher Buckingham

If you have not completed the first two parts of building an expert system, you can download a suggested lisp program from BlackBoard that was given in the last tutorial. Note that the forward chaining only works on the given starting facts (it can't find additional facts from anywhere else) so you will need to think how the facts should be set up to test your program. My example starts with the following facts

```
(setf *facts '((give-milk y) (hair y)(chew-cud y)(black-stripes y)))
```

because they will lead to the identification of a zebra after some forward chaining. The current explanation is pretty poor, though, because it simply outputs the starting facts and the rule that proved the final goal:

```
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).
```

1. Try running the program with different starting facts: no facts; facts with a goal in it already; facts that will not lead to a goal.
2. Study the outputs and see if you can understand how the program generated them. There are two main problems with the current program:
   (a) if the forward chaining can't find a solution, it has no way of accessing external facts;
   (b) if a solution is found, the explanation does not show how the rules were executed, only the last one.

This lab will first try to generate a better explanation of the answer using backward chaining; if you have time, the final part will have functions that ask the user for additional facts and then forward chain with these.

1 Explaining the answer using backward chaining

When a goal has been found or the forward chaining has gone as far as it can without asking for additional facts, the system needs to provide the new information and explain how it was obtained. One approach is to use backward chaining where the system tries to show a chain of rules from the goal back to the starting data. Backward chaining is much better at providing explanations because it is more focused than forward chaining.

Using our example, backward chaining will say that the zebra has black stripes, is an ungulate because it is a mammal and has hoofs, and is a mammal because it it has hair and gives milk. This explanation is pieced together by the list of rules leading backwards from the goal rule found by forward chaining to the starting facts.

2 Create a function called backward-chaining

It will be a recursive function, so you need to read the accompanying lisp notes in this lab folder on BlackBoard. Your backward-chaining function works on a list of rules, where the list starts off with the final rule returned by the forward-chaining function. Backward chaining then adds rules to the rule list where those rules are used to prove the first rule. The function call is as follows:

```
> (backward-chain rule-chain *rules)
```

where rule-chain will be a list containing a goal rule such as

```
((ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))
```

and the function will return the list of rules leading from the goal rule back to the initial facts that were used to prove the goal rule by forward chaining.

The backward chaining algorithm can be tested using some test variables and called on a goal rule, as follows (with comments added):

```
;; starting facts
> *facts
((GIVE-MILK Y) (HAIR Y) (CHEW-CUD Y) (BLACK-STRIPES Y))

> (setf rule-chain '((ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y)))))

> (backward-chain rule-chain *rules)

rule chain: ((ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

rule chain: ((UNGULATE ((MAMMAL Y) (HOOFS Y)))(ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

rule chain: ((MAMMAL ((HAIR Y) (GIVE-MILK Y)))(UNGULATE ((MAMMAL Y)(HOOFS Y)))
             (ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

;; returns the final rule-chain
((MAMMAL ((HAIR Y) (GIVE-MILK Y))) (UNGULATE ((MAMMAL Y) (HOOFS Y)))
 (ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))
```

2.1 Your task: create the backward-chaining function

The following steps for the function should make it easier for you to produce your first recursive definition:

1. Set up the function definition parameters
2. get the conditions of the first rule in the chain
3. go through the list of conditions
4. if the condition is also a rule conclusion
   (a) add the rule to the rule chain
   (b) backward chain on the new rule chain (with the new rule added to it) and set the rule chain to the result.
5. Return the result

When it comes to testing your rule, try putting in `format` statements to see what the value of the rule chain is on each recursive call of the function:

```
(format t "~%rule chain: ~a~%" rule-chain)
```

which would produce the following output if it was in the rule and run as shown earlier:

```
> (backward-chain (list goal) *rules)

rule chain: ((ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

rule chain: ((UNGULATE ((MAMMAL Y) (HOOFS Y)))(ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

rule chain: ((MAMMAL ((HAIR Y) (GIVE-MILK Y)))(UNGULATE ((MAMMAL Y) (HOOFS Y)))
             (ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))

((MAMMAL ((HAIR Y) (GIVE-MILK Y))) (UNGULATE ((MAMMAL Y) (HOOFS Y)))
 (ZEBRA ((UNGULATE Y) (BLACK-STRIPES Y))))
```

where the last list is what the function returned.

3 Using the list of rules to explain the proof

1. Write a function that uses the list of rules output by the backward-chaining algorithm to explain the proof to the user.
2. The function will use a dolist to go through the rule list and format statements to provide some useful explanation from the chain of rules.