Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **3** Due: **9/29/98** 6.034 Item # **3**

Be sure to indicate on your solution the time, room number and recitation instructor for the recitation section that you normally attend. Also, write BOTH partners' names on the solution.

Goals

- You continue your introduction to rule-based systems, this time using backward chaining.
- You explore some of the design issues in backward chaining, to explore its strengths and weaknesses.
- You get some practice in distinguishing between knowledge level and implementation level issues when discussing programs.

Background

Files mentioned in this problem set can be found in `/mit/6.034/ps3/` under Athena (remember to `setup 6.034`) or through our Web page.

As in the previous problem set, we have a simple Scheme program that functions as a rule interpreter, but this one is a backward chainer. In order to help you see the similarities and differences in the two techniques, we use the same creature identification rules you've seen used in the textbook and in class, but run them backward.

The code is contained in the files `match.scm`, `unify.scm` and `back.scm`. You can load the rule interpreter and sample rule systems by doing `(load "/mit/6.034/ps3/load.scm")`.

As before, the `match.scm` file defines the function `match` and `do-match` and auxiliaries, while `unify.scm` handles unification (matching variables). The `back.scm` file defines the rule interpreter. (One warning to those with a sense of, or access to, history: this backward chainer is different in important ways from ones used in the class previously.)

A quick note on terminology: the IF parts of a rule are variously called its antecedents or premises; the THEN parts are referred to as consequents or conclusions.

Here are a few important top-level functions for this interpreter:

- `(backchain <pattern>)` – activates the interpreter with the current rules and assertions, searching for a way to establish the `<pattern>`. It has a 3-step process:
  1. see if the pattern is already in the assertions,
  2. if not, see if there is a rule whose conclusion matches the pattern, in which case, try to establish the premises of that rule;
  3. if there is no pattern in the database and no relevant rule, ask the user.
- `(clear-rules)`, `(remember-rules <rule1> ... <rule n>)`, `(clear-assertions)`, `(remember-assertion <assert>)`, and `(display-assertions)` all work as in the previous problem set.
- `(make-variables-unique)` – if we are chaining back we may run into two rules in a chain that happen to have variables with the same name. This procedure gives all the variables newly manfactured names to ensure there is no conflict.

Problem 1

Part A

Examine the file `zoo.scm` (a copy is at the end of this handout); it contains the creature identification rules from the textbook. To familiarize yourself with backward chaining and with the rules, draw the tree that would sprout as a result of trying to determine what kind of animal Tony is, i.e., all the possible subgoals that would arise from

```
(backchain '(Tony is a (? x)))
```

(For reasons that will become clear, don't try to run the code to do this yet. Draw the tree by hand for now; you'll need it for the rest of the problem set.)

The beginning of that tree is shown below, so you have a clear idea of what kind of thing to draw.

This will give you a picture of the overall knowledge base. If you don't recall the backward chaining process from class, hand-simulate the code for one cycle and it will become clearer.

Part B

Run the backchainer to determine whether Tony is specifically a tiger, by giving it the command

```
(backchain '(Tony is a tiger))
```

and using the following facts (which the system will ask you about):

```
(Tony has hair): (n)
(Tony gives milk): (y)
(Tony eats meat): (u)
(Tony has pointed teeth): (y)
(Tony has claws): (y)
(Tony has forward-pointing eyes): (y)
(Tony has tawny color): (y)
(Tony has black stripes): (y)
```

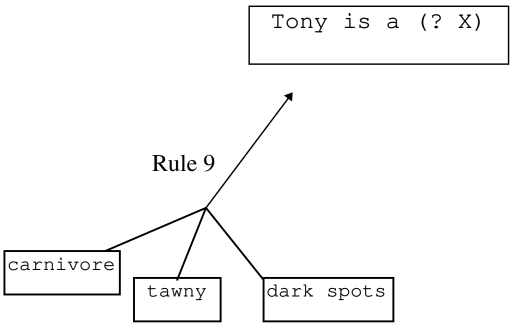
Note two important things about the queston-asking routine:

1. An *implementation-level fact*: it reads lists, so you need to reply to a question with (y), (n), or (u) including the parentheses.
2. A *knowledge-level fact*: you can indicate that you know an assertion to be true by saying (y), that you know the assertion to be false by saying (n), or that you have no knowledge about the assertion, by indicating (u), for unknown. Hence when you reply (u) in the following exchange:

   ```
   Have no assertion or successful rule to determine whether (Tony eats meat).
   Is this true [y or n]? (u)
   ```

   you are not saying he doesn't eat meet, merely that you don't know one way or the other. (Our code treats (u) and (n) equivalently, essentially using them to mean "not known to be true," and hence of no use in establishing a subgoal.)

Annotate the tree you drew in the previous part of the problem to indicate the path the backchainer followed in doing this example.


Tony is a (? X)
Rule 9
carnivore
tawny
dark spots

Problem 2

This problem looks at additional implementation-level issues.

Part A

In the code you will find the following segment in `try-rule`:

```
    (cond((no? result)
;;;        (format #t (indentit "[Rule ~a is irrelevant to ~a.]")
;;;        (rule-name rule) hypothesis)
;;;         uncomment the line above for extra tracing info
           '())
```

Explain what the code means by saying that a rule is irrelevant to the hypothesis. You may wish to uncomment the lines and look at a few examples.

Part B

You may have noticed in running the (Tony is a tiger) example that the system asks more questions than it should. Indicate the source of this unnecessary work by describing in a few sentences what the program is failing to do.

Repair the problem with a small change to the code; hand in a listing of the procedure you revised.

Problem 3

This problem goes back to the knowledge level to explore what the zoo rules "know" and how they represent that knowledge.

Part A

Ask the system to determine what animal Tony is, by having it do

```
(backchain '(Tony is a (? x)))
```

Use the following facts:

```
(Tony has hair): (u)
(Tony gives milk): (y)
(Tony eats meat): (u)
(Tony has pointed teeth): (n)
(Tony has hoofs): (y)
(Tony has a long neck): (y)
(Tony has long legs): (y)
(Tony has dark spots): (y)
```

You would of course expect this to be a giraffe. What does the system infer instead?

Part B

Why does the system do this? That is, explain in a few words why you think the system behaves as it does. Note that the problem has nothing to do with the code; this is a knowledge-level issue. What's wrong with what the system knows?

[You may have noticed that re-ordering the rules will have an effect on the problem, but it won't really solve it. The core problem is in the knowledge content, not in the interpreter.]

Part C

Reword the appropriate rules to fix the problem you described above. Re-run the example in the first part of this exercise to show that the problem is indeed fixed. Turn in a listing of your revised rules, and a trace of them working on the example above.

A Copy of the Zoo Rules

```
(define (make-zoo-rules)
  (clear-rules)
  (remember-rules
   '(Z1 IF
        ((? animal) has hair)
        THEN ((? animal) is a mammal))
   '(Z2 IF
        ((? animal) gives milk)
        THEN ((? animal) is a mammal))
   '(Z3 IF
        ((? animal) has feathers)
        THEN ((? animal) is a bird))
   '(Z4 IF
        ((? animal) flies)
        ((? animal) lays eggs)
        THEN ((? animal) is a bird))
   '(Z5 IF
        ((? animal) is a mammal)
        ((? animal) eats meat)
        THEN ((? animal) is a carnivore))
   '(Z6 IF
        ((? animal) is a mammal)
        ((? animal) has pointed teeth)
        ((? animal) has claws)
        ((? animal) has forward-pointing eyes)
        THEN ((? animal) is a carnivore))
   '(Z7 IF
        ((? animal) is a mammal)
        ((? animal) has hoofs)
        THEN ((? animal) is a ungulate))
   '(Z8 IF
        ((? animal) is a mammal)
        ((? animal) chews cud)
        THEN ((? animal) is a ungulate))
   '(Z9 IF
        ((? animal) is a carnivore)
        ((? animal) has tawny color)
        ((? animal) has dark spots)
        THEN ((? animal) is a cheetah))
   '(Z10 IF
         ((? animal) is a carnivore)
         ((? animal) has tawny color)
         ((? animal) has black stripes)
         THEN ((? animal) is a tiger))
   '(Z11 IF
         ((? animal) is a ungulate)
         ((? animal) has long neck)
         ((? animal) has long legs)
         ((? animal) has dark spots)
         THEN ((? animal) is a giraffe))
   '(Z12 IF
         ((? animal) is a ungulate)
         ((? animal) has black stripes)
         THEN ((? animal) is a zebra))
   '(Z13 IF
         ((? animal) is a bird)
```

```
       ((? animal) does not fly)
       ((? animal) has long legs)
       ((? animal) has long neck)
       ((? animal) is black and white)
       THEN ((? animal) is a ostrich))
 '(Z14 IF
       ((? animal) is a bird)
       ((? animal) does not fly)
       ((? animal) swims)
       ((? animal) is black and white)
       THEN ((? animal) is a penguin))
 '(Z15 IF
       ((? animal) is a bird)
       ((? animal) flies well)
       THEN ((? animal) is a albatross)))
(format #t "~%Zookeeper rules recorded."))
```