Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **2** Due: **9/22/96** 6.034 Item # **2**

Goals

- You are introduced to rule-based systems using forward chaining.
- You explore some of the design issues in forward chaining systems.

The problems in this problem set are (nearly) independent of each other; the last two problems are weakly related.

Background

Files mentioned in this problem set can be found in `/mit/6.034/ps2/` under Athena (remember to `setup 6.034`) or through our Web page.

In this problem set, we begin our study of rule-based systems. To assist us in this, we will introduce a very simple Scheme program that interprets forward-chaining rule systems analogous to those in the textbook (see handout). The code for this system is contained in the files `match.scm` and `forward.scm`. First do `(cd "/mit/6.034/ps2")` or to whatever directory holds the files. You can then load the rule interpreter and sample rule systems, by doing `(load "load.scm")`.

The `match.scm` file defines the function `match` and `do-match` and auxiliaries. This is the unmodified matcher from Problem Set 1; it does not support segment variables. In principle, you could load your modified matcher and then you could write rules with segment variables. But, we will not do that here.

The `forward.scm` file defines the rule interpreter. Basically, the system keeps a global list of rules (bound to `*rules*`) and a global list of assertions (bound to `*assertions*`). The function call `(chain)` begins the cycle of looking for rules that are triggered, firing the rules and repeating until no rules are triggered. The following are the few important top-level functions for this interpreter:

- `(chain)` – activates the interpreter with the current rules and assertions. Finds the first triggered rule each time through the execution cycle. The first instance of this rule is fired.
- `(clear-rules)` – removes all current rules.
- `(remember-rules <rule1> ... <rule n>)` – add the rules to set of current rules.
- `(clear-assertions)` – removes all current assertions.
- `(remember-assertion <assertion>)` – add the assertion to the current assertions.
- `(display-assertions)` – show the current list of assertions.

The interpreter first identifies all the rules whose antecedents match assertions in the current set, whose AND-IF tests are not false and which either adds a new assertion or deletes some existing assertion. These are the triggered rules. Then, it calls the function choose-rule-instance to select a rule instance to be fired. choose-rule-instance currently just picks the first instance in the list. Note that the triggered rules are in the order that they were added to the rule set. The chosen rule is fired: new assertions are added, existing assertions are deleted, and EVALUATING and SAYING clauses are evaluated. Then the cycle is repeated until no rules are triggered. If the assertion (STOP) is added to data base, rule chaining stops.

Problem 1

Let's familiarize ourselves with the operation of the interpreter.

Part A

As the interpreter runs it prints out the names of all the triggered rules. For example, (bagger-run) produces, among other things:

```
 Triggered rules: (B9 B9 B9 B9 B9 B9)
Rule B9 adds (BAG3 IS CURRENT-BAG)
Rule B9 deletes (BAG2 IS CURRENT-BAG)
Rule B9 deletes (BAG3 IS AVAILABLE-BAG)
Start a fresh bag.
```

Note that there are generally multiple copies of a single rule listed there, B9 in this case. Explain (2 or 3 sentences) why this is so in terms of the matching process in forward.scm. Make sure to indicate how the order of the rules and assertions affects the process. Use an example to illustrate your answer.

Part B

Explain (1 or 2 sentences) how the implementation we have used for computing and selecting rule instances is wasteful.

Part C

Assume we are using chain to call the rule interpreter. Give an algebraic expression that shows how the **worst case** running time to fire a single rule grows as a function of the number of assertions, call it S, and the number of rules, call it R. Assume that the rules are simple IF-THEN rules with k, the worst case number of antecededents per rule.

Justify your answer by indicating which Scheme functions in the interpreter contribute what factors to your answer. Be brief.

`Hint: You should find the answer horrifying!`

Problem 2

Consider the following trivial facts about the operation of a car that are useful in suggesting repairs:

- For the headlights to work one needs a charged battery.
- To rotate the engine, one needs a charged battery and a working starter motor.
- To start the engine one needs fuel, a working ignition and an engine that rotates.

Write a small set of rules that concludes with one of the following facts:

```
(repair is nothing)
(repair is charge-battery)
(repair is fix-starter-motor)
(repair is fix-ignition)
(repair is add-fuel)
```

The input is a set of assertions that describe the current state of the car. The initial assertions are drawn from:

```
(engine-starts yes)   (engine-starts no)
(engine-rotates yes)  (engine-rotates no)
(headlights-work yes) (headlights-work no)
(fuel yes)            (fuel no)
```

Of course, you are not limited to have only these assertions in your rules; your rules are free to produce intermediate assertions on their way to the final repair assertion.

Hand in your rules and the output when run on the test cases found in `/mit/6.034/ps2/auto.scm`. Add your rules in the place indicated in this file and then execute `(auto-run)`.

You do not need to check for impossible conditions, e.g. `(engine-starts yes)` and `(engine-rotates no)`.

Part A

One could imagine coding the knowledge in this simple problem using rules such as:

```
(R1 IF (headlights-work yes)
    THEN (battery-charged yes))
(R2 IF (engine-rotates yes)
    THEN (battery-charged yes)
         (starter-motor-works yes))
```

Explain briefly what problems you would face if you encode the knowledge this way.

Problem 3

Note that the rule interpreter does not "understand" the content of the assertions. So, for example, in the auto example above it does not know that exactly one of `(engine-starts yes)` and `(engine-starts no)` can be true. Furthermore, the interpreter does not encode the usual rules of logic. It does not, for example, automatically know how to handle facts related by AND, OR or NOT.

Consider the following "rules":

- If the coin comes up heads, I win.
- If the coin comes up tails, you lose.

It follows that you always lose, i.e. don't win. Clearly, if you simply encode the rules:

```
(R1 IF HEADS THEN (WIN ME))
(R2 IF TAILS THEN (LOSE YOU))
```

you haven't achieved anything. You need to encode the relationship between ME and YOU and winning and losing.

```
(R3 IF (WIN ME) THEN (LOSE YOU))
(R4 IF (LOSE ME) THEN (WIN YOU))
(R5 IF (WIN YOU) THEN (LOSE ME))
(R6 IF (LOSE YOU) THEN (WIN ME))
```

But you also need to know the relationship between Head and Tails, at this point, you need to start introducing NOT, to say that exactly one of Heads or Tails can be true. But, then you have to codify how to deal with NOT, etc. Congratulations, you have just re-invented propositional logic...

It is easy to encode some of the rules of propositional logic in the rule interpreter. See `/mit/6.034/ps2/logic.scm` for one such encoding. You don't need to understand the details of these rules at this point. Suffice it to say that all these rules encode valid inferences. If one believes the antecedents of the rule, then one should believe the consequent. One needs most[1] of these rules to guarantee one can in principle complete any proof succesfully.

Part A

Run the logic example `(logic-run)` given in `/mit/6.034/ps2/logic.scm`. Describe what happens. Don't hand in a printout of the result.

Part B

Many of the logic rules are "explosive". They produce a consequent that is a new assertion but which matches the antecedent of the rule. Here is an example of an imaginary explosive rule:

[1] This is neither a minimal nor a complete set of rules.

```
'(R1 IF (?  P) THEN ((?  P)))
```

This rule simply makes a list of whatever pattern it matches. So if the assertion `hello` exists in the database, R1 will trigger and match `P` to `hello`. The assertion `(hello)` is then added to the database. R1 again triggers, this time matching `P` to `(hello)`. The assertion `((hello))` is then added to the database. R1 again triggers, this time matching `P` to `((hello))`. The assertion `(((hello)))` is then added to the database... This is the sense in which we mean that R1 is "explosive".

Which of the rules in `/mit/6.034/ps2/logic.scm` are of the "explosive" type. Think about what these rules are doing, can we do without them?

Part C

Describe (1 or 2 sentences) the problem in using these rules together with a rule-order conflict-resolution strategy.

Problem 4

Let's consider implementing a new conflict-resolution strategy, different from strict rule-order.

Part A

Let's modify the interpreter so as make the behavior of the rule interpreter less dependent on the order of the rules. In particular, implement a simple form of **recency**-based conflict resolution, that is, fire the least recently fired rule among the triggered rules.

In order to implement this, you will need some way of keeping track of the most recent "step" of the rule interpreter that each rule was fired. We suggest you use a **hash-table** for this. A hash-table is an efficient method for keeping an association between two sets of items, known as the keys and the data. In this case the name of the rules will be the keys and the step number of their most recent firing will be the data. In MIT Scheme (pp 151–161 of reference manual), the following functions are provided:

- `(make-eq-hash-table)` – creates and returns an empty hash table.
- `(hash-table/put! <hash-table> <key> <datum>)` associates datum with key in the table.
- `(hash-table/get <hash-table> <key> <default>)` returns the datum associated with the specified key. If the key does not have an association, the specified default is returned.

To get these functions you must include `(load-option 'hash-table)` at the top of your .scm file. For example:

```
(load-option 'hash-table)
(set! table (make-eq-hash-table))
(hash-table/put! table 'a 1)    ; creates association between 'a and 1
(hash-table/get table 'a 0) -> 1
```

You will also need to sort the rules based on recency, recall that MIT Scheme has a built-in sort function: (sort <list> <comparison-function>), e.g. (sort '(3 4 2) <) -> (2 3 4).

Hand in any modified function definitions from forward.scm as well as any new code required to implement this feature.

Part B

Show the modified interpreter code running on the logic example (logic-run) given in /mit/6.034/ps2/logic.scm.

Part C

Is this recency-based conflict resolution a good idea for this rule set? Have you done away with the explosiveness of the logic rules by using the recency-based conflict resolution? Explain (1 or 2 sentences).