Alec Goebel
AI
11/14/07

RI DESIGN

The overall architecture of the rules engine was based upon the forward chaining defined in Winston's book. The RI consists of the interpreter and all of its helper functions, in addition to bindings, facts, and rules. Rules themselves are made up of conditionals and actions as well as bindings. There also is the grid and node classes from Project 0 which have been left more or less untouched.

The algorithm for the RI is fairly simplistic:

```
For each rule
    Check if any facts fit into the rules bindings
    Check if both bindings and condition evaluates to true
        If so, add to conflict set
    Sort conflict set by priority
    Sort conflict set by condition weight (manually set in analyze cond)
    Select first rule and run it (but do not remove)
Repeat until empty rule list, stop, or empty conflictset
```

There were a few other noteworthy algorithms. The process-cond macro was an attempt to translate conditions into natural language. This allowed for non-scheme syntax to be used to define conditions and it lead to very legible code. Actions were not treated similarly due to their complex nature and specific functions needed to be created either way.

The other strange algorithm is the bind macro. Bind takes a list of (name condition) and then does the equivalent of a let*. Each pair is translated to the name and then a processed condition which takes the current variable name as well as all previous. For example:

```
(bind (?test1 (?test1 exists))
      (?test2 ((?test2 exists) and (not (?test2 same-as ?test1)))))
```

Evaluates to:

```
(list  (cons  '?test1  (lambda (?test1) (…)))
       (cons '?test 2 (lambda (?test1 ?test2)  (…)))
```

Both the rules and facts were stored in a list, becoming a list of lists. Facts were a simple pair of symbols/numbers. Rules, on the other hand, were complicated lists. Rules contained their name, two template lambdas (evaluated at runtime) for storing the condition and the actions triggered. After the changes needed for Task 2, it also needed to store bindings which are a list of pairs of names and conditions.

The main assumption I made with the RI is that the user would be comfortable in a scheme like environment. In any sort of problem, a great deal of helper functions would need to be made to represent the new actions and tests the engine must perform. Along those lines I decided the syntax should resemble CLIPS and the actions hard coded.

There are several important global variables. The main two are 'facts' and 'rules', the working memory. Every time defrule or deffacts get called, the content of the two gets modified. There are also 'initfacts' and 'initrules' which are a copy of the WM when initialize is called.

The only other two global variables of note are the debug and running booleans. These simply store the state. When they RI is running (not stopped) running is true and if the debug output should be printed, debug must be set to true.

The language itself is, again, based upon CLIPS syntax. The declaration of a rule is made by defrule and facts are created by deffact. There is a specific order to where things may go in defrule and it is no longer anywhere near the CLIPS standard in all but the most general sense. Conditions also have a very specific format, but almost every possibility is covered, and those which are not can be worked around. For the most part I feel the conditions are intuitive to use and easy to read.

An IF THEN wrapper for defrule is there in case it was desired.

The test cases for Task 1 and Task 2 are in the main directory.

The problem I selected for task two was an implementation of the noisy neighbors logic puzzle. The domain is configuration. A real world example of the situation would be to map a landscape based on limited data fed by each vehicle in the field. In this scenario, almost none of them know where they are and know very little about who is where or the heights of the land around them.

Since forward chaining is designed to fan in, I started with a list of facts which contained every possibility of who was on which square and at what height. I then used rules based on the information provided by the vehicles to begin weeding out options.

The sub-steps were every time the RI discovered that it had solved one of the squares height or vehicle. At that point it cleared that option from all the other squares, marked the square as done, added the data to the landscape, and continued on its way. It also knew how to recognize when it had completed its task by checking the landscape for holes.

The rules themselves were as follows (they needed to be expanded):

- The Person in Vehicle B sees an elevation of 8 to the north
- The Person in Vehicle C sees an elevation of 7 to the south, 6 on either the east or west, and only has two people next to him
- The Person in Vehicle D is sandwiched between 2 and 5 on his east or west side and south of A
- The Person in Vehicle E is east or west of A and south of 9
- The Person in Vehicle F is south of E and is standing at 5
- The Person in Vehicle G is east or west of 1 and south of 4
- The Person in Vehicle I is north of 3 and west of 8 and west of h
- 3 is south of 6 but north of 1

Most bindings took the form of ?cell where ?cell contains some value n. This was used to represent “the person in vehicle x.” Then, using the bindings, most conditions took the form of (not ?cell south-of value) to check if the value wasn’t in the south. These “next-to” functions simply check if the cell is on the grid and if it has the value given. Lastly, the actions mostly took the form of overwriting the current value of n. Each rule could be read as “find cell with value n and if it doesn’t meet the requirements to have value n, remove n.”