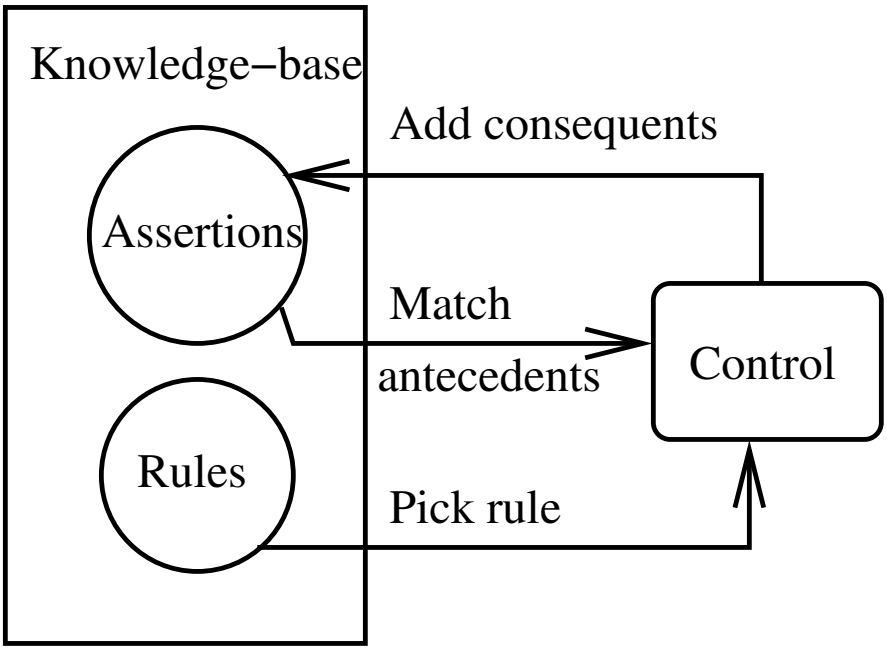
Notes on Rule/Knowledge-Based Systems [1]

Rule-based Systems

This section discusses the most important aspects of rule-based systems.

Separation of *Knowledge* and *Control*

One of the most important aspects of rule-based systems is the separation of the knowledge (assertions + rules), from the control (procedures by which we find new conclusions from what we currently know). The control can be implemented via *forward* or *backward chaining*. The rules help us *add* new assertions to the working memory in *deduction systems*, or *add and delete* assertions in *production systems*. Here is a figure that illustrates this point.



Why and How

Recall that by using a *goal (and/or) tree*, rule-based systems can answer questions about their behavior. In the ZOOKEEPER example, one may ask the questions: Why did you show that Stretch was an ungulate? How did you show it?

Some Limitations

One limitation of rule-based systems is that they cannot conclude new assertions or handle questions outside their domain of expertise. Humans, on the other hand, can use their experience to transfer what they know about a problem to solve other similar problems.

[1] Original date: September 24, 2004; Last updated: November 5, 2006.

Another limitation of the type of rule-based systems presented in this course is that they cannot handle uncertainty. For instance, in some cases we might want to state: "If A then *often* B." Rule-based systems have been extended to try to handle such cases by using some numeric value quantifying the *strength* of the rules. This is sometimes given in terms of a probability value.

Forward Chaining

ASSERTIONS ⇒ CONCLUSIONS

The pseudocode that follows is for a naive implementation of forward chaining.

```
FORWARD-CHAINING
repeat
  for every rule do
    if antecedents match assertions in the working memory and consequents would
    change the working memory then
      Create triggered rule instance
    end if
  end for
  Pick one triggered rule instance, using conflict resolution strategy if needed, and fire it
  (throw away other instances)
until no change in working memory, or no STOP signal
```

Note that this is a general implementation that applies to both deduction and production/reaction systems. Be aware that many variations are possible. For instance, in deduction systems, forward chaining *usually* fires a rule (instance) as soon as it is triggered. Note also that even in deduction systems forward chaining might use a conflict resolution strategy to decide the order in which it considers the rules for matching (in the for-loop).

Backward Chaining

CONCLUSIONS (HYPOTHESES) ⇒ ASSERTIONS

The pseudocode that follows is for a naive implementation of backward chaining.

```
BACKWARD-CHAINING(H)
if H matches an assertion in working memory then
  return true
end if
if there is no rule with a consequent that matches H then
  ASK USER or ASSUME false
end if
for every rule R with a consequent that matches H do
  if for all antecedents A of rule R, we have BACKWARD-CHAINING(A) = true then
    return true
  end if
end for
return false
```

Note the recursive nature of backward chaining. Note also that the for-loop creates the *or-nodes* of the goal (and/or) tree, while the if-statement inside that for-loop creates the *and-nodes*.

Other more efficient implementations are possible. For instance, we might add assertion to the working memory for later reuse as we find them to be true during the backward chaining process. Also, many variants resulting from different implementation decisions are possible. For example, the system can ask the user even after all rules with a consequent that match the hypothesis fail. Another option is to use conflict resolution to decide the order in which the system will consider the rules with consequents that match the hypothesis (in the for-loop).