

Query System Described by Diagrams

Basic Pattern Matching

The query system works by matching patterns with elements inside a given database whose elements are defined by the command *(assert! <rule or datum>)*. One way to describe this occurrence is by passing in a frame with pre-bound variables and creating new frames based on the matches of the given pattern.

Say the following facts were put into our database:

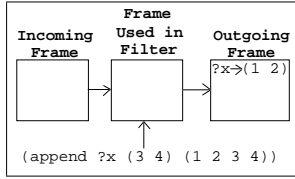
```
(assert! (rule (append () ?x ?x)))
(assert! (rule (append (?u . ?v) ?y (?u . ?w))
              (append ?v ?y ?w)))
```

These rules makes the database act as though we have defined all possible append elements. This saves us time since we don't have to type in the infinite possibilities and also makes the database search faster since we can use *unification* in pattern matching.

Now we wanted to find the following pattern in the database:

```
(append ?x (3 4) (1 2 3 4))
```

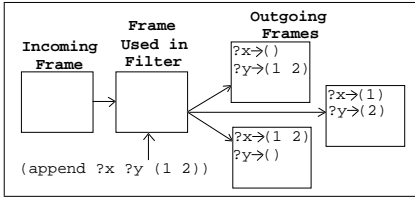
The diagram to describe this query is shown to the right. The incoming frame has no bound variables because queries always start with empty frames. In the filter region, the filter goes through every database entry, using previously bound variables (none in this case), and returns all frames with bound variables that match the given query pattern. In this case the only match is for *?x* to be bound with *(1 2)*.



How about a case that had multiple results? Like the query:

```
(append ?x ?y (1 2))
```

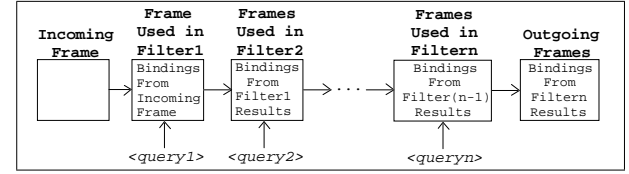
Notice how it makes a frame for each matching pattern. All of these frames represent matches and are passed on to the user (which displays the answers) or to another filter.



The AND Query Operator

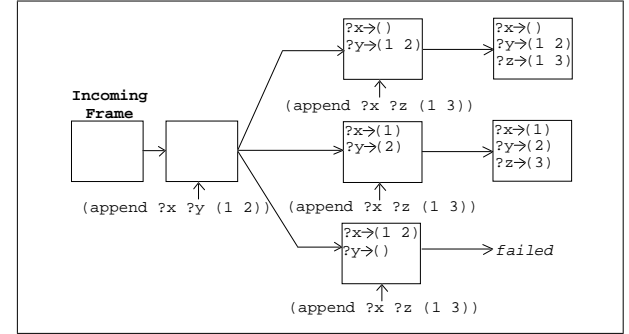
The *and* operator in the Query System performs sequential filters. Here's how the following expression would be drawn:

```
(and <query1> <query2> ... <queryn>)
```



An example of using this in an actual query could be:

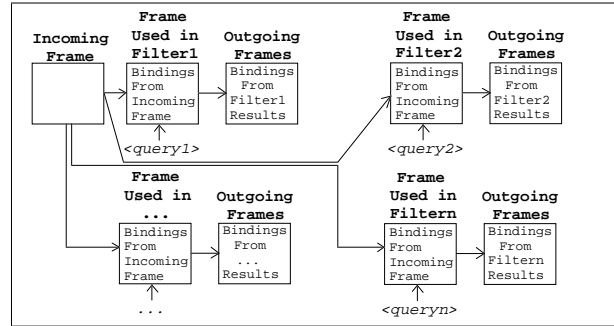
```
(and (append ?x ?y (1 2))
      (append ?x ?z (1 3)))
```



The OR Query Operator

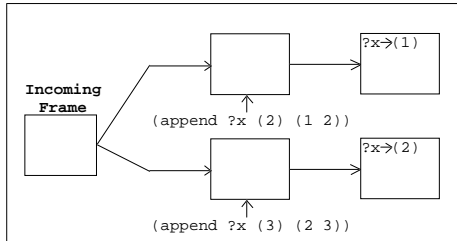
The *or* operator in the Query System performs parallel filters. Here's how the following expression would be drawn:

```
(or <query1> <query2> ... <queryn>)
```



An example of using this in an actual query could be:

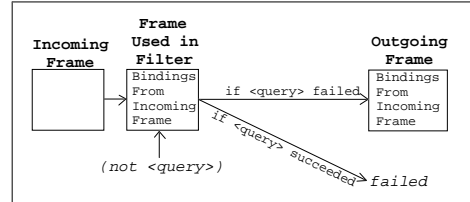
```
(or (append ?x (2) (1 2))
     (append ?x (3) (2 3)))
```



The NOT Query Operator

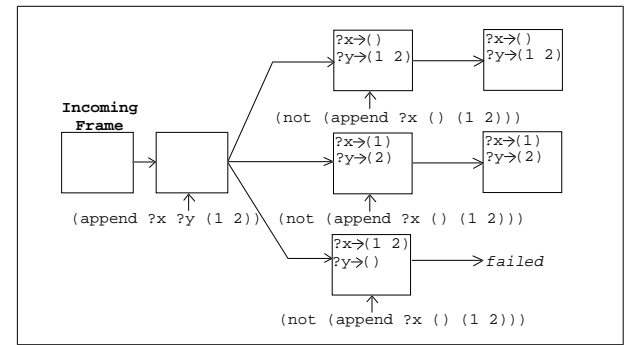
The *not* operator takes the given bound variable information and fails if the pattern matches something inside the database. If the pattern doesn't match something inside the database, the *not* operator returns the frame unchanged (no newly bound variables):

```
(not <query>)
```



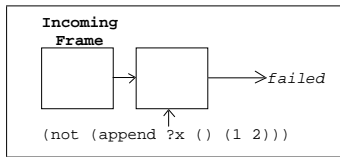
An example of using the *not* operator would be:

```
(and (append ?x ?y (1 2))
      (not (append ?x () (1 2))))
```



Another example of using the *not* operator where we encounter the problem discussed in the book would be:

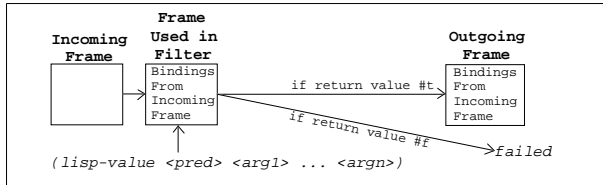
```
(and (not (append ?x () (1 2)))
      (append ?x ?y (1 2)))
```



The LISP-VALUE Query Operator

The *lisp-value* query operator is similar to the *not* operator and applies the given predicate procedure to the given arguments. If the result is *#t*, the outgoing frame is the same as the incoming frame. If the result is *#f*, there is no outgoing frame and *failed* is returned. A *lisp-value* query would be drawn in the same way as a *not* operator would, except that if one of the variables used in it is undefined, the query produces an error (which causes our implementation to exit out to Scheme):

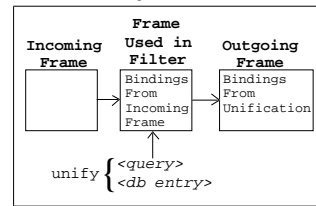
```
(lisp-value <pred> <arg.> ... <arg.>)
```



A Normal Database Match¹

Normal Database Matches go through the following steps:

1. Unify the rule with the query and current frame.
 - a. If unification fails, the match fails.²
2. Match succeeds and returns the resulting frame.

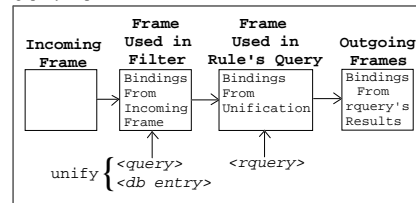


Matching Rules in the Query System

Rules are database entries that describe a pattern of possible entries in the database. What determines whether the rules match or not is a correspondence with the rule's pattern followed by an optional matching with a query. The lack of the query in a rule causes the query to succeed as long as the rule's pattern successfully unifies with the query.

(rule <pattern> <query>) - This defines a particular *pattern* that, if unifiable, will return one or more matching query responses, given that the *query* also matches to one or more items in the database.

(rule <pattern>) - This defines a particular *pattern* that, if unifiable, will return a matching query response.



Notice the keen resemblance to the AND query operator. The difference here is that it's done during the satisfaction of a query, not in the making of a query.

¹ It's normal in that it's not matching to a rule.

² Our query system actually uses a subset of unification in this case, pattern matching, but the effects are the same.

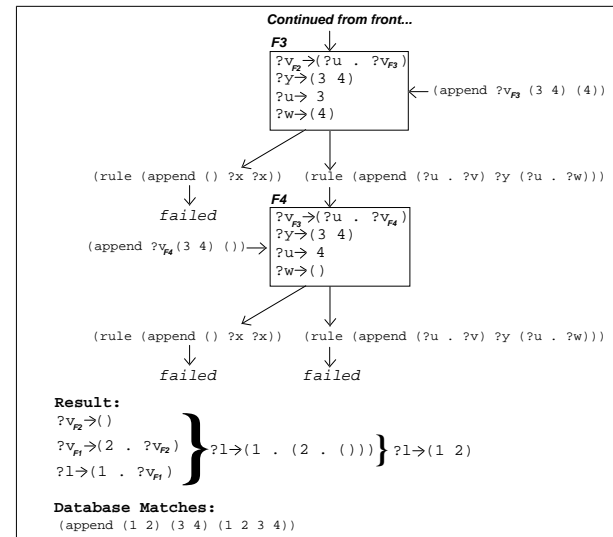
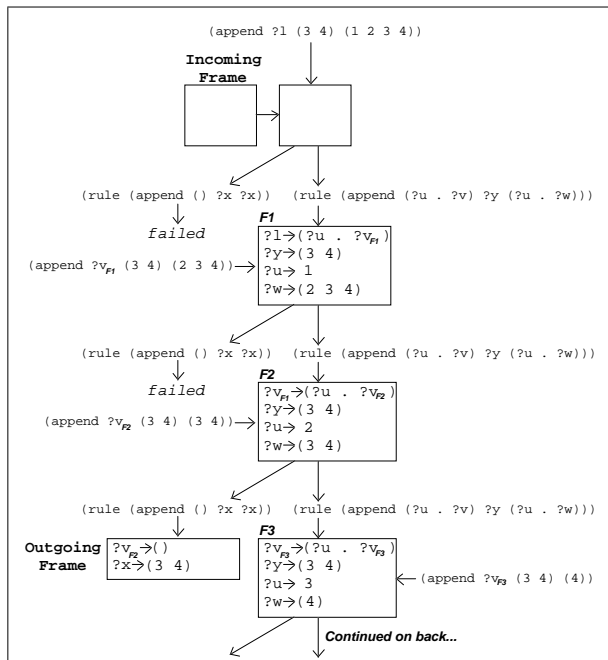
Tracing the Append Rule

As an example of tracing rules, I'm going to use the *append* rules I have been using throughout this document.

```
(assert! (rule (append () ?x ?x)))
(assert! (rule (append (?u . ?v) ?y (?u . ?w))
              (append ?v ?y ?w)))
```

A simple query would be something like:

```
(append ?l (3 4) (1 2 3 4))
```



Making Rules Like Append

Just by looking at the *append* rule, it isn't necessarily obvious how such a rule was conceived. The first step to writing a rule like *append* is to make a list of the simplest cases of *append*:

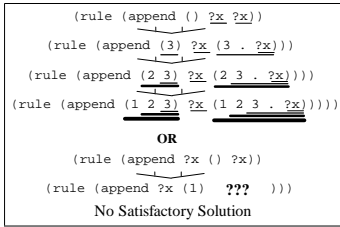
```
(append () () ()) | (append () (1) (1)) | (append (4) () (4))
(append () (1 2) (1 2)) | (append (3 4) () (3 4))
```

Now we see that there are two possibilities for *base case*³ rules. It's not yet obvious which one would be useful:

```
(rule (append () ?x ?x)) or (rule (append ?x () ?x))
```

³ This lower case seems to be a good starting place for making these types of rules. I'm wondering if it could be given the name *obvious case* or *obvious rule*.

Now using our two possible *base* rules, we need to see what more general cases would look like by slowly making our base cases more complicated.



Notice that one base rule lead to a dead end. The other looks promising because we could easily build larger cases from the *base case*. Now what we need is a rule that will unbuild a large general case and result in the *base case*!

By looking at the way the cases are reduced, we can see that our rule will have a form:

```
(rule (append (?u . ?v) ?x (?u . ?w)) ...)
```

This can be seen by noticing that at each step, we don't need to break down the middle element, and only need to break the 1st and 3rd elements into a pair. Now the query part of the rule is what narrows down to the *base case*, so we write it so that it will ask for the more narrow case.

```
(rule (append (?u . ?v) ?x (?u . ?w))
      (append ?v ?x ?w))
```

Now we have all of the rules necessary for defining all cases of appending one list to another! Let's repeat them one last time, just so you can see our final result:

```
(assert! (rule (append () ?x ?x)))
(assert! (rule (append (?u . ?v) ?y (?u . ?w))
              (append ?v ?y ?w)))
```