

The Query System

Pattern Matching

There are three elements used in pattern matching, the pattern to match, the datum being checked, and the frame of currently bound variables. In our *query system*, our data is represented using lists, so our *pattern matcher* looks for lists that follow a given pattern. The return value from the *pattern matcher* is a frame with the variables bound to their matching data, or an indication of failure.

Here are some examples of patterns:

- (?x) - This looks for a list containing *one* element.
- (?x ?y) - This looks for a list containing *two* elements that can be the same or different.
- (?x ?x) - This looks for a list containing *two* elements that are the same.
- (?x . ?y) - This looks for a list containing an element in the car of the dotted pair, and another element in the cdr of the dotted pair.
- (5 . ?x) - This looks for a pair with a 5 in its car and anything else in the cdr.

Pattern Matching Using Unification

Unification is the matching of one pattern to another pattern so that they both make a pattern consistent to both. The process of *unification* depends on what elements are being unified. If the elements are incapable of being made equal, the two patterns cannot be *unified*. Things are unified as follows:

1. If the patterns are *atoms*¹ and are equal, the two patterns are already unified.
2. If one of the patterns is a *variable*, the *variable* must be matched to the other pattern.
3. If both patterns are pairs, then the car of both patterns are unified, and then the cdrs of both pairs are unified.
4. Anything else is a failure to be unified.

Matching variables to other variables and patterns can be somewhat tricky. It also follows a set of rules:

1. If the other pattern is the same *variable*, they are already matched.
2. If the *variable* is unbound, then it must be bound to the other pattern².
3. If the *variable* is bound, then what it's bound to must be unified with the other pattern.
4. Anything else is a failure to be matched.

¹ Atoms in this case is anything that isn't a pair.
² Note that the other pattern can be a variable.

As an example, we can try unifying (?x ?x) and ((a ?y c) (a b ?z)).

STEP 1:	(?x ?x)	((a ?y c) (a b ?z))	
	<i>a pair</i>	<i>a pair</i>	
	Both patterns are pairs, so first unify the car and then follow that with unifying the cdr.		
STEP 2:	?x	(a ?y c)	?x => (a ?y c)
	<i>a variable</i>	<i>a pair</i>	
	The first pattern is a variable, and is not bound to any pattern yet, so we bind it to the given pattern. (Note that this is not a recursive definition.)		
STEP 3:	(?x)	((a b ?z))	?x => (a ?y c)
	<i>a pair</i>	<i>a pair</i>	
	Finished unifying the car, we move to the cdr. The cdr consists of pairs. This means we must first unify the car and then follow that with unifying the cdr.		
STEP 4:	?x	(a b ?z)	?x => (a ?y c)
	<i>a pair</i>	<i>a pair</i>	
	The first pattern is a variable, and is bound to a pattern already, so the pattern to which it is bound must be unified with the other pattern.		
STEP 5:	(a ?y c)	(a b ?z)	?x => (a ?y c)
	<i>a pair</i>	<i>a pair</i>	
	Both patterns are pairs, so first unify the car and then follow that with unifying the cdr.		
STEP 6:	a	a	?x => (a ?y c)
	<i>an atom</i>	<i>an atom</i>	
	Both patterns are atoms, and since they are equal, they are already unified.		
STEP 7:	(?y c)	(b ?z)	?x => (a ?y c)
	<i>a pair</i>	<i>a pair</i>	
	Finished unifying the car, we move to the cdr. The cdr consists of pairs. This means we must first unify the car and then follow that with unifying the cdr.		
STEP 8:	?y	b	?x => (a ?y c)
	<i>a variable</i>	<i>an atom</i>	?y => b
	The first pattern is a variable, and is not bound to any pattern yet, so we bind it to the given pattern.		
STEP 9:	(c)	(?z)	?x => (a ?y c)
	<i>a pair</i>	<i>a pair</i>	?y => b
	Finished unifying the car, we move to the cdr. The cdr consists of pairs. This means we must first unify the car and then follow that with unifying the cdr.		
STEP 10:	c	?z	?x => (a ?y c)
	<i>an atom</i>	<i>a variable</i>	?y => b
			?z => c
	The second pattern is a variable, and is not bound to any pattern yet, so we bind it to the given pattern.		

STEP 11:	()	()	?x => (a ?y c)
	<i>an atom</i>	<i>an atom</i>	?y => b
			?z => c
	Finished unifying the car, we move to the cdr. The cdr consists of atoms. Since the atoms are equal, they are already unified.		
STEP 12:	()	()	?x => (a ?y c)
	<i>an atom</i>	<i>an atom</i>	?y => b
			?z => c
	Finished unifying the pattern ?x was bound to with the other pattern, we also finish unifying the car of the pair and move to the cdr. The cdr consists of atoms. Since the atoms are equal, they are already unified.		
CONCLUSION:	(?x ?x) and ((a ?y c) (a b ?z))	?x => (a ?y c)	
	both unify to:	?y => b	
	((a b c) (a b c))	?z => c	

Assert! and Rule

In the query system, we are given the capability of either asserting something to be added into the database, or query for some information from the database. By default the system assumes we are asking for information, so in order to tell it that we want to store something, we need to use the *assert!* special form:

(assert! <rule or datum>) - Stores the rule or datum in the database for informing future queries.

To easily define generalizations, the query system has the ability to define rules which describe data which can be inferred from other data. Rules can be defined in two forms:

- (rule <conclusion> <body>) - This defines a particular *conclusion* that can be drawn given the query that makes up the *body*.
- (rule <conclusion>) - This defines a particular *conclusion* that is always true given any values of the variables inside the *conclusion*³.

SICP's complex example of how rules can be used is the description of *append*.

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

The first rule is the identity case (which ends up being similar to a base case) for appending two lists. Anything appended with an empty list is itself. The second rule says that we can say

³ The absence of a query is always a successful query.

(?u . ?v) appends with ?y to form (?u . ?z) given that (*append* ?v ?y ?z) is true. This rule defines the very essence of what *append* does. The elements at the front of the first list are put at the front of the result, and the rest of the elements in the first list appended to the second list make up the rest.

And & Or

In our query system, there is a way for us to make sure many conditions all apply. This is done with the *and* special form. The *and* special form evaluates each query in its body sequentially, and if one fails, it fails to match.

(and <query₁> <query₂> ... <query_n>)

To make sure that at least one condition applies, there is an *or* special form. The *or* special form evaluates each query in its body simultaneously (in parallel), and unless they all fail, it doesn't fail to the match.

(or <query₁> <query₂> ... <query_n>)

Not & Lisp-Value

Two peculiar special forms in the query system are *not* and *lisp-value*. Both of these require that the variables be bound with values before they are used, or else the query may have strange results (or an error).

The *not* special form excludes only the unacceptable data of the query. The problem with having unbound variables is that anything can fill that space, so everything that follows the given pattern will be rejected⁴

(not <query₁>)

The *lisp-value* special form is more limited than the *not* special form in that it requires the variables all be bound before it can function. If any of the variables are unbound it could generate an error (our implementation does this), and abort the evaluation of the given query.

(lisp-value <predicate> <arg₁> ... <arg_n>)

⁴ This may be what you want, so don't reject it as an illegitimate use for the *not* special form.