Example 2

The second example considers the specification of rules and the development of functions to apply them. Some student texts simplify rule application by using rules that have no matching capability. Using a common example rules may be written...

(rule 32 (has fido hair) => (isa fido mammal))

The preconditions of this kind of rule are tested for equality with known facts which means that such a rule could conclude nothing given the fact (has lassie hair). This is an over simplification which can hide important issues in the design of expert systems and other rule-based inference engines. Using the matcher allows more realistic rules to be developed - the rule above would be written...

(rule 32 (has ?x hair) => (isa ?x mammal))

This example will work towards writing a rule application mechanism & a small inference engine which deals with matching rules but before considering rules with matching capabilities we will consider writing a simplified rule-apply function which does not use matching.

Using the information (antecedents & consequents) contained in the following rule...

```
(rule 32 (has fido hair) => (isa fido mammal))
```

...and the set of facts held in data below...

```
(defparameter data
  '((has tweetie feathers)
    (has fido hair)
    (goes daisy moo)
    ))
```

we can build a function which uses antecedents & consequents explicitly

```
(defun r-app1 (antecedent consequent facts)
  (if (member antecedent facts :test #'equal)
      (cons consequent facts)
    facts
    ))


> (r-app1 '(has fido hair) '(isa fido mammal) data)
==> ((isa fido mammal) (has tweetie feathers)
     (has fido hair) (goes daisy moo))
```

we can use a more declarative (& better) rule structure as in first example if we pull it apart with *mlet* (you can check out examples of *mlet* in the matcher documentation)..

```
(defun r-app2 (rule facts)
  (mlet ('(rule ?n ?antecedent => ?consequent) rule)
        (if (member #?antecedent facts :test #'equal)
            (cons #?consequent facts)
          facts
          )))
```

```
> (r-app2 '(rule 32 (goes daisy moo) => (isa daisy cow))
           data)
==> ((isa daisy cow) (has tweetie feathers)
     (has fido hair) (goes daisy moo))
```

multiple antecedents & consequents can be handled by using set-subset $< and set-union $+ functions (these functions are provided in utils.cl)

```
(defparameter data
  '((has tweetie feathers)
    (has fido hair)
    (goes tweetie tweet)    ; new fact
    (goes daisy moo)
    ))

(defun r-app3 (rule facts)
  (mlet ('(rule ?n ??ante => ??conseq) rule)
        (if ($< #?ante facts)
            ($+ #?conseq facts)
          facts
          )))

> (r-app3 '(rule bird1
                 (goes tweetie tweet) (has tweetie feathers)
                 => (isa tweetie bird))
          data)
==> ((isa tweetie bird) (has tweetie feathers) (has fido hair)
     (goes tweetie tweet) (goes daisy moo))
```

We can now take the example further by considering rules which have matching capabilities. With matching rules application can be achieved directly using the matcher's forevery form...

```
 (setf family
 '((parent Sarah Tom)  (parent Steve Joe)  (parent Sally Sam)
   (parent Ellen Sarah) (parent Emma Bill) (parent Rob  Sally)))

; applying...
;    ((parent ?a ?b)(parent ?b ?c)) => (grandparent ?a ?c)

> (forevery ('((parent-of ?a ?b) (parent-of ?b ?c)) family)
      (match>> '(grandparent ?a ?c)))
→  ((grandparent ellen tom) (grandparent rob sam))
```

This approach can be used to develop a general purpose mechanism for applying rules to facts which returns an updated set of facts. This function uses the matcher to deconstruct a rule and then forevery to apply it. Notice that the mechanism for rule deconstruction and the details of repeated rule application are all removed from the programmer who is left to focus on rule antecedents, consequents and facts.

```
(defun apply-rule (r facts)
 (mlet ('(rule ?n ??antecedents => ??consequents) r)
   (forevery (#?antecedents facts)
     (setf facts ($+ (match>> #?consequents) facts)))
   facts))
```

```
> (apply-rule
    '(rule 15 (parent ?a ?b) (parent ?b ?c) => (grandparent ?a ?c))
    family)

→ ((grandparent rob sam)  (grandparent ellen tom)
    (parent sarah tom)  (parent ellen sarah)  (parent steve joe)....)
```

The example is completed with a mechanism which repeatedly applies a set of rules to update facts, continuing until the rules are unable to generate any new inference - acting as a forward chaining process. It is only at this final stage that the program code explicitly uses any repetitive construct - in this case an iterative form. Some Lisp programmers may question the structure of the iteration in the function preferring to replace **let** and **loop** with **do** or make more use of the loop macro facilities. However the function is presented as shown since this does not require students to know any details of the loop macro or work with Lisps do/do* forms since some students report that the unusual structure of do/do* is hard to read.

```
(defun fwd-chain (rules facts)
  (let (old-facts
    (loop (setf old-facts facts)
          (setf facts
            (reduce #'$+
               (mapcar #'(lambda (r) (apply-rule r facts)) rules)))
          (if ($= old-facts facts) (return facts))
        )))

; another set of facts & rules

(defvar facts1
 '((big elephant) (small mouse) (small sparrow) (big whale)
   (on elephant mouse)))

(defvar rules1
 '((rule 1 (heavy ?x)(small ?y)(on ?x ?y) => (squashed ?y) (sad ?x))
   (rule 2 (big ?x)   => (heavy ?x))
   (rule 3 (light ?x) => (portable ?x))
   (rule 4 (small ?x) => (light ?x))  ))

> (fwd-chain rules1 facts1))
→ ((portable sparrow) (portable mouse) (squashed mouse)
    (sad elephant)  (heavy whale) (heavy elephant)
    (light sparrow) (light mouse) (big elephant) (small mouse)
    (small sparrow) (big whale)   (on elephant mouse))
```