Example 1

This example examines how a generalised query mechanism can be built to retrieve information from a sets of statements where each statements is a triple of the form: (relation object value).

For the sake of experimentation this example presents a simple world environment describing a collection of blocks. In Lisp this can be defined as...

```
(defvar *blocks*
 '((isa b1 cube) (isa b2 wedge) (isa  b3 cube) (isa  b4  wedge)
   (isa b5 cube) (isa b6 wedge) (color b1 red) (color  b2  red)
   (color  b3  red) (color  b4  blue) (color  b5  blue)
   (color b6 blue)(on b1 table) (on b2 table) (on b5 table)))
```

The matcher's *foreach* form can be used to retrieve the names of objects satisfying a specified relation from this type of structure. The form below matches all triple statements of the *isa-wedge* type and returns the relevant object names.

```
>(foreach ('(isa ?obj wedge) *blocks*)  #?obj) → (b2 b4 b6)
```

quote & backquote

There are two forms of quote, the normal one which causes what it precedes to be treated as data (ie: it is not evaluated), so...

```
> '(hello banana (+ 4 5) spam) → (hello banana (+ 4 5) spam)
> '(first '(a b c)) → (first '(a b c))
```

Sometimes we want to selectively evaluate parts of quoted data. To do this we use backquote and precede what we want evaluated by a comma...

```
> `(hello banana ,(+ 4 5) spam) → (hello banana 9 spam)
> `(blah ,(rest '(a b c)) spam) → (blah (b c) spam)
```

If you want to *list* an evaluated result into data use comma and @

```
> `(blah ,@(rest '(a b c)) spam) → (blah b c spam)
```

This approach can be used to build a general purpose function lookup which is defined and used as follows...

```
(defun lookup (pair tuples)
 (foreach(`(,(first pair) ?x ,(second pair)) tuples)
    #?x))

> (lookup '(isa cube) *blocks*) → (b1 b3 b5)
> (lookup '(on table) *blocks*) → (b1 b2 b5)
```

sets & set operators

Sets as unordered collections of *things* – normally data and most often instances of some class or in Lisp they may be collections of symbols, tuples, etc. Sets do not contain duplicates. Most modern languages support sets and various utilities to handle set operations.

The utils.lsp collection from the www.agent-domain.org site contains set utilities and there is a some information there explaining how to use them. For our purposes here we can think of sets as being lists of symbols (in no important order). Set operators allow us to find the *intersection* of sets or their *union*, etc.

For example, set intersection...

```
> ($* '(cat dog rat frog) '(fish poodle cat dog)) → (dog cat)
```

Set union...

```
> ($+ '(cat dog rat frog) '(fish poodle cat dog))
     → (frog rat fish poodle cat dog)
```

Lookup returns a set of object names so results of different lookup operations can be combined with set operators like $* (set intersection) and $+ (set union). This allows multiple queries to be satisfied.

```
> ($* (lookup '(isa cube) *blocks*)(lookup '(on table)*blocks*))
→ (b5 b1)
```

reduce

The set intersection function $* can only be applied to two sets at a time. In order apply this function (and lots of others) to multiple arguments we can use reduce.

```
> (reduce #'+ '(23 45 576 78 90)) → 812
> (reduce #'+ '(23 4 5 8 9)) → 49
> (reduce #'- '(23 4 5 8 9)) → -3
> (trace +) → (+)
> (reduce #'+ '(23 4 5 8 9))
 0[4]: (+ 23 4)
 0[4]: returned 27
 0[4]: (+ 27 5)
 0[4]: returned 32
 0[4]: (+ 32 8)
 0[4]: returned 40
 0[4]: (+ 40 9)
 0[4]: returned 49
49
> (reduce #'$* '((a b c d e) (a b x r) (a b d op))) → (a b)
```

mapcar

mapcar is a mapping function which repeatedly applies a function to a list of data & collects the results. Note the use of #' to pass a function object as a piece of data

```
> (mapcar #'1+ '(1 3 7 123)) → (2 4 8 124)
> (mapcar #'first '((c a t)(d o g)(r a t)(f r o g))) → (c d r f)
> (mapcar #'third '((c a t)(d o g)(r a t)(f r o g))) → (t g t o)
```

lambda

lambda allows us to specify new functions in-line and build functions without having to give them a name

```
> (setf add2 #'(lambda (x) (+ x 2)))
#<Interpreted Function (unnamed) @ #x20d09d3a>

> (funcall add2 5) → 7

> ((lambda (x) (+ x 7)) 20) → 27
```

one of the main uses for lambda is to specify functions for one-off uses, often in calls to mapping functions like mapcar

```
> (mapcar #'(lambda (x) (+ x 7)) '(12 2 70 120)) → (19 9 77 127)
```

A generalised query function can be built which maps pairs like (isa cube) and (color blue) over the lookup function and reduces results using $* (in the case of logically ANDed queries).

```
(defun query-and (pairs triples)
  (reduce #'$*  (mapcar #'(lambda (p) (lookup p triples)) pairs)))

> (query-and '((isa cube)(on table)) *blocks*) → (b5 b1)
```

more on passing functions as data

It is possible (and useful sometimes) to store functions in variables. Here are some examples of how this could work...

```
(defun modify (fn num)
  (funcall fn num))

(setf fns (list #'1+ #'sqrt #'(lambda (x) (* x x))))

> (modify (first fns) 16) → 17
> (modify (second fns) 16) → 4.0
> (modify (third fns) 16) → 256
```

Logically ORed queries can be similarly handled using $+ in place of $*. The final query function shown below can then be used in conjunction with two variables Qand & Qor (introduced for readibility) which deal with ANDed results and ORed results.

```
(defvar Qand #'$*)    (defvar Qor  #'$+)

(defun query (logic-op pairs triples)
  (reduce logic-op  (mapcar #'(lambda (p) (lookup p triples)) pairs)))

> (query Qand '((isa cube)(color red)) *blocks*) → (b3 b1)
> (query Qor '((isa cube)(color red)) *blocks*) → (b5 b1 b2 b3)
```

finished program

```
(defun lookup (pair tuples)
 (foreach(`(,(first pair) ?x ,(second pair)) tuples)
    #?x))

(defvar Qand #'$*)    (defvar Qor  #'$+)

(defun query (logic-op pairs triples)
  (reduce logic-op  (mapcar #'(lambda (p) (lookup p triples)) pairs)))
```