```
who= cs310 3[ab] Adam_Minter
here= /home/aminter1/Project-3-CS310/310.3
total 32
4 drwxr-xr-x  2 aminter1 aminter1 4096 2012-04-01 06:20 .
4 drwxr-xr-x 10 aminter1 aminter1 4096 2012-03-30 03:08 ..
8 -rw-r--r--  1 aminter1 aminter1 5773 2012-04-01 06:20 15.lisp
4 -rw-r--r--  1 aminter1 aminter1 1134 2012-04-01 03:57 answers3a.txt
4 -rw-r--r--  1 aminter1 aminter1 1115 2012-04-01 05:46 answers3b.txt
4 -rw-r--r--  1 aminter1 aminter1  273 2012-04-01 05:32 answers3c.txt
4 -rw-r--r--  1 aminter1 aminter1  238 2012-03-10 00:12 main.lisp


--------------------------------------
running ...

;testing  !RANDS
;testing  !TIME-IT
;testing  !ZERO
;testing  !ONE
;testing  !AGE
293
;testing  !OLDEST
; pass : 5 = 100.0%
; fail : 0 =   0.0%
NIL




=====|  15.lisp |============================
#| ######################################################################

Paul Graham's chapter 15 (see http://goo.gl/1OCc5) from Ansi COMMON
LISP (see http://www.paulgraham.com/acl.html) implements a minimal
Prolog system. It can't do everything that Prolog can do but, heh,
what do you expect of 100 lines of code.

###################################################################### |#

(defun match (x y &optional binds)
  (cond
   ((eql x y)        (values binds t))
   ((assoc x binds)  (match (binding x binds) y binds))
   ((assoc y binds)  (match x (binding y binds) binds))
   ((var? x)         (values (cons (cons x y) binds) t))
   ((var? y)         (values (cons (cons y x) binds) t))
   (t
    (when (and (consp x) (consp y))
      (multiple-value-bind (b2 yes)
                           (match (car x) (car y) binds)
        (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))

(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))

(defun prove (expr &optional binds)
  (case (car expr)
    ; cs310 students! insert new code here
     (do          (prove-code (second expr) binds) (list binds))
```



```
     (say         (prove-code expr binds) (list binds))
     (>           (prove-operator (car expr) (cdr expr) binds))
     (<           (prove-operator (car expr) (cdr expr) binds))
     (and         (prove-and (reverse (cdr expr)) binds))
     (or          (prove-or  (cdr expr) binds))
     (not         (prove-not (cadr expr) binds))
     (t           (prove-simple (car expr) (cdr expr) binds))))



(defun prove-code (expr binds)
  (labels ((lets (binds want)
             (mapcar #'(lambda (x) `(,x ',(binding x binds))) want)))
    (let* ((vars (lets binds (vars-in expr)))
           (code `(let ,vars ,expr)))
      (eval code))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r)
                                          binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr))
             (vars-in (cdr expr)))))


(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))

(defun prove-operator (operator args binds)
  (when (prove-code (cons operator args) binds) (list binds)))

;(defun prove-is (arg1 arg2 binds)
;  (prove-code (match arg1 arg2 binds))(list binds))

(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         (declare (ignorable ,@(vars-in query)))
         ,@body))))

(defun data0 ()
  (clrhash *rules*)
```

```
    (<- (parent donald nancy))
    (<- (parent donald debbie))
    (<- (male donald))
    (<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
    (<- (= ?x ?x))
    (<- (sibling ?x ?y) (and (parent ?z ?x)
                             (parent ?z ?y)
                             (not (= ?x ?y)))))

(defun data1 ()
  (clrhash *rules*) ; must start with this
  (<- (= ?x ?x))
  (<- (person matt  m  23  40000))
  (<- (person dean  m  90  90000))
  (<- (person clint m 100 100000000))
  (<- (person marge f  80 100000000))
  (<- (younger ?x ?y) (and (person ?x ?g1   ?age1 ?salary1)
                           (do (format t "~a~%" ?x))  ; <== needs "do"
                           (person ?y ?g2 ?age2)
                           (is ?factor (/ 11 10))
                           (not (= ?x ?y))
                           (< (* ?age1 ?factor) ?age2 ?salary2) ; <== needs "<"
                           (say "my isn't ~a too young" ?x))) ; <== needs say
  (<- (sameSex ?x ?y) (and (person ?x ?gender ?a1)
                           (person ?y ?gender ?a2)))
  (<- (hates   ?x ?y) (and (sameSex ?x ?y) (younger ?y ?x)))
  (<- (oldest ?person) (and (person ?person ?gender ?age ?salary)
                            (not (and (person ?personb ?genderb ?ageb ?salaryb)
                                      (> ?ageb ?age))))))
)

(deftest !zero ()
  (data0)
  (test
      "DONALD is the father of DEBBIE
       DONALD is the father of NANCY
       DEBBIE is the sibling of NANCY.
       NANCY is the sibling of DEBBIE.
       "
      (with-output-to-string (s)
        (with-answer (father ?x ?y)
          (format s "~A is the father of ~A~%" ?x ?y))
        (with-answer (sibling ?x ?y)
          (format s "~A is the sibling of ~A.~%" ?x ?y)))))

(deftest !one ()
  (data1)
  (with-output-to-string (s)
    (with-answer (hates ?x ?y)
      (format s "~a hates ~a~%" ?x ?y))))

(deftest !age ()
  (data-ages)
  (let ((x 0))
    (with-answer (age ?a)
      (incf x ?a))
    (format t "~a" x)
    (test 293 x)))

(defun data-ages ()
  (clrhash *rules*)
  (<- (person matt  m  23  40000))
  (<- (person dean  m  90  90000))
  (<- (person clint m 100 100000000))
  (<- (person marge f  80 100000000))
  (<- (age ?a) (and (person ?x ?gender ?a ?salary1))))

(deftest !oldest ()
  (data1)
  (test "CLINT is the oldest person"
        (with-output-to-string (s)
          (with-answer (oldest ?x)
            (format s "~a~% is the oldest person" ?x)))))
```



```
=====|  answers3a.txt |============================
1. (match '?x 1 nil) returns the following:
((?X . 1))
T

2. Why does the match function return two values?
The first return is a list of associated lists that show each match found. The
second return is a flag T or nil, indicating whether or not the match was
successful.

3. Give an example for the kind of structure that would satisfy var.
?adam_minter

4. Give an example for the kind of structure that would not satisfy var.
adam_minter

5. Why does binding call itself recursively before it looks at the cdr of the
current binding?
This function has to be recursive, becasue matching can build up binding lists
in which a variable is only indirectly associated with it's value: ?x might be
bound to an in virtue of the list containing both (?x . ?y) and (?y . a).
(p. 250)

6. The function data1 includes a predicate 'sameSex' that matches to two
persons. Except for gender, every other field has a suffix that is either
'1' or '2'. Does gender need a suffix? Why or why not?
Gender can only have 2 different values, male or female. There isn't any other
option, so there doesn't need to be any extra organization to the rule.



=====|  answers3b.txt |============================
1. Why does prove-simple use mapcan as the list mapping operator?
Hint: "nil".
A failed proof returns nil, otherwise, mapcan returns all the lists, and
doesn't return the nils.

2.  The following functions all return lists of multiple bindings:
prove-simple, prove-and, prove-or, prove-not.
Why don't we just return one binding?
Because there are often times multiple bindings. Only returning one binding
would, for instance, return only one of the parents. Returning multiple
bindings will return both bindings containing both parents in question.

3. Given an example of "r" before and after it is processed by change-vars.
Each expression is rehashed using the value passed in for r.

4. The function prove-code evals a structure called code that creates a
let statement which defines some vars (local variables) then calls some
code expr. What is going on here?
The let creates the list of bindings that hold each variable, code creates
the code block that declares each binding, inserting the expression inside
each. Prove-code returns the entire expression with each binding set in an
environment.



=====|  answers3c.txt |============================
1. In âM-^@M-^X15.lispâM-^@M-^Y, the equality test is handled a funny way: (<- (
= ?x ?x)).
Is that a typo? Should not the second "?x" be different to the first?
And should there be a body to the rule?
It's not a typo. A variable is always equal to itself, so it needs to stay as
is.



=====|  main.lisp |============================
(handler-bind ((style-warning #'muffle-warning))
  (mapc 'load '(
                "../tricks.lisp"
```

```
                    "15.lisp"
                    )))

(defun ! () (load "main.lisp"))

(defun main ()
  (tests))


(defun hello (&optional (who "world"))
       (format nil "hello ~a~%" who))
```