```
;Michael Amann
;CSE 240
;Lab 4 Richard Whitehouse
;TTH @ 3:15-4:30

;Write a Lisp function power-of-two which takes a number as a parameter
;and returns the nth powerof 2.
;         (power-of-two '8)                             ==> 256

;Pre-conditions: The argument list will only accept a positive number
;Post-conditions: The number two is multiplied by the argument as a
;power of two.

(defun power-of-two (x)
      (if (numberp x)
            (cond ((equal x '0) 1)
            (t (* 2 (power-of-two(- x 1))))
            )
      'IMPROPER_ARGUMENT_LIST
      )
)

;Write a Lisp function replicate which takes two arguments, the first
;an expression and the second a non-negative integer. It returns a list
;containing the expression
;copied the given number of times.

;    (replicate 'a 4)                             ==> (a a a a)
;    (replicate '(magic) 0)                       ==> nil
;    (replicate '(1 2) 3)                         ==> ((1 2) (1 2) (1 2))

;Pre-conditions: The functions takes two arguments, an expression such
;as 'a or 'nil, and a number.
;Post-conditions: The functions returns a list containing the number of
;instances of that expression which is determined by the second
;argument.

(defun replicate(x y)
      (if (and (and x y)
               (and (numberp y) (or (> y  0) (equal y 0))))
                 (cond ((zerop y) nil)
                   ((equal x nil)nil)
                       (t (cons x(replicate x (- y 1))))
                  )
      'IMPROPER_ARGUMENT_LIST
      )
)


;Write a Lisp function non-nil which takes a list as a parameter,
;and returns a transformed version of the list such that all nil
;elements are changed to 0 and all non-nil elements are changed to 1.

;    (non-nil '(a nil (b) (nil) 2))                  ==> (1 0 1 1 1)
```

```
;Pre-conditions: The function requires a list of atoms.
;Post-conditions: The function returns a list containing 1's and zeros.
;The zero's replace and nil element, and all other elements are
;replaced by 1's.

(defun non-nil (x)
      (if (listp x)
            (cond ((equal x nil) nil)
                  ((equal (first x) nil) (cons '0 (non-nil (rest x))))
                  ((not (equal (first x)nil))(cons '1 (non-nil (rest
x))))
            )
            'IMPROPER_ARGUMENT_LIST
      )
)


;Write a Lisp function count-atoms that counts all the atoms in a list
;passed as the parameter.
;        (count-atoms '(a b c d))                     ==> 4
;        (count-atoms '(a (b c (d e) f) (g h)))       ==> 8
;        (count-atoms '(a (b c) d))                   ==> 4

;Pre-conditions: The functions requires a list of atoms and or lists.
;Post-conditions: The functions traverses the binary tree data
;structure and counts all the atoms in the list. It then returns a
;number indicating how many atoms were
;contained in that list.

(defun count-atoms (x)
      (if (listp x)
            (cond ((null x) 0)
                  ((atom (first x)) (+ 1 (count-atoms (rest x))))
                  ((listp (first x)) (+ (count-atoms (first x)) (count-
atoms (rest x))))
            )
            'X_MUST_BE_A_LIST
      )
)


;Write a Lisp function flatten which returns a list of all the atoms in
;x. The argument x can be an atom of a list whose components, can be
;atoms or lists.
;        (flatten '(a (b (c d)) e))                    ==> (a b c d e)

;Pre-conditions: The function requires a list containing atoms and/or
;lists
;Post-conditions: The function returns a list of all the atoms in the
;list, including any atoms in a list.

(defun flatten (x)
      (if  (listp x)
            (cond ((equal x  nil) nil)
                  ((atom (first x)) (cons (first x) (flatten (rest
x))))
```

```
                  ((listp (first x)) (append  (flatten (first x))
(flatten (rest x))))
            )
      'X_MUST_BE_A_LIST
      )
 )

;Write a Lisp function my_member that works the same as the lisp member
;function (do not use the member function in your solution).

;Pre-conditions: The function takes a list of elements and a test case
;argument. The argument list must consist of an atom for x and a list
;for y. Post-conditions: The function compares the test case element to
;the list and searches for a match of the test case. If none is found,
;nil is returned. If a match is found, that element along with the
;remaining elements after the match are returned.

(defun my_member (x y)
      (if (and (atom x) (listp y))
            (cond       ((equal y  nil) nil)
                  ((and (atom (first y)) (equal (first y) x))
                        (cons (first y)  (rest y)))
                  ((listp (first y))(my_member x (rest y)))
                  ((and (atom (first y))(not(equal(first y) x)))
                        (my_member x (rest y)))
            )
      'IMPROPER_ARGUMENT_LIST
         )
      )

;Write a Lisp function sub-splice that takes three parameters: the new
;item, the old item to be changed and the list to be edited. Your
;function will return a new version of the list with all the
;occurrences of the old item replaced with the new item.

 ;(sub-splice 3 1 '(1 2 (1 2 (1 2))))            ==> (3 2 (3 2 (3 2)))
 ;(sub-splice '(1 2) 'b '(a b c))                ==> (a 1 2 c)
 ;(sub-splice '(1 2) 'b '(a (b c) d))            ==> (a (1 2 c) d)

;Pre-conditions: The function takes three parameters. They are the new
;item, the old item to be changed and the list to be edited. The first
;parameter can be a list or an atom. The second parameter must be an
;atom. The third parameter must be a list. Post-conditions: The
;function returns a new version of the list to be changed with all the
;occurrences of the old item replaced with the new item.

(defun sub-splice (x y z)
      (if   (or (and (atom x)(atom y)(listp z))
            (and (listp x)(atom y)(listp z)))
      (cond ((equal z nil) nil)
            ((and (atom (first z))(not(equal(first z)y)))
                  (cons (first z)(sub-splice x y (rest z))))
            ((and (atom (first z)) (equal (first z) y)(listp x))
                  (cons (first x) (cons (first(rest x)) (sub-splice x y
(rest z)))))
            ((and (atom (first z)) (equal (first z)y))
                  (cons x (sub-splice x y (rest z))))
            ((listp (first z)) (cons (sub-splice x y (first z))
                              (sub-splice x y (rest z))))

      )
      'NIL
      )
)
```

```
/*Michael Amann
  CSE 240 TTh@ 3:14-4:30
  Richard Whitehouse
  Prolog Lab3 */


/*define the following relations on lists:

   1.delete(A, X, B) true if list B is the result of deleting a single
occurence of X from
   list A, ie: delete([3, 1, 4, 2], 4, [3, 1, 2]).

   Delete takes three parameters, a list, the element to be excluded and
the resulting list
   or an unbound variable. Passing the list to the append function, it
is broken in two
   excluding the element supplied that is to be removed. Somelist is
given the head of the
   list, and A is given the rest. Then Somelist and the rest of the list
are appended to
   form a new list less the given element.*/

      delete(A, X, B):-
            append(Somelist,[X|Xs], A),
                  append(Somelist, Xs, B).


/* 2.sorts(A, B)is true if list B is an ordered permutation of list A.
      Sample call
      ?- sorts([a,b,c,d],[b,d,c,a]).
      yes */

/* Sorts takes two parameters, either two lists or a list an an unbound
variable. The base
   case is two empty lists. Sorts first breaks up the first list into
Head and Tail (or X,Xs).
   Sorts then passes the rest of the first list and an unbound variable
Ys back to the relation.
   X is then deleted from Ys1 and the remaining list is placed in the
empty list Ys and returned
   */

      sorts([], []).
      sorts([X|Xs], Ys1):-
      sorts(Xs, Ys),
            select(X, Ys1, Ys).

/*Select(X,Xs,Ys) is true if Ys is the result of removing the first
occurence of X from Xs.
  Select accomplishes this task by first taking the supplied element,X,
and comparing it
  with the head of the supplied list. If X and the head match, the rest
of the list is passed
  to the unbound variable and the function returns. If X and the head do
not match, the list
  is passed as a formal parameter and broken up with the head being
saved to Y. Y is then
  placed at the head of the unbound variable and the rest of Y, Ys is
passed back to the
  function along with X and the rest of the unbound variable. When X and
```

```
the head finally do
  match, the rest of the list is saved to Xs, and the recursive calls
build the list back up
  excluding the selected element.*/

            select(X,[X|Xs],Xs).
                  select(X, [Y|Ys],[Y|Zs]):-
                  select(X, Ys, Zs).


/* 3.substitute(X, Y, L1, L2) where L2 is the result of substituting Y
for all occurrences of X
    in L1:
     substitute(a, x, [a,b,a,c], [x,b,x,c]) is true
     substitute(a, x, [a,b,a,c], [a,b,x,c]) is false */

 /* substitute(X,Y,Xs,Ys) is true if the list Ys is the result of
substituting Y for all
    occurrences of X in the list Xs.
    Substitute completes its task by continually comparing the heads of
two list to see if
    they meet the requirements of the relation. To begin, substitute
breaks up two lists and
    assings the heads to X and Y. The rest of the two lists are passed
back into the function.
    The heads of those list are then stored in Z and compared to X. If X
matches Z then the
    substitution is not complete and the relation fails. If there is no
match, the list is
    parsed through until the base case of the empty list is found, and
upon returning, the list
    is rebuilt and returned.*/

      substitute(X,Y,[],[]).
      substitute(X,Y,[X|Xs],[Y|Ys]):- substitute(X,Y,Xs,Ys).
      substitute(X,Y,[Z|Xs],[Z|Ys]):- X \==Z, substitute(X,Y,Xs,Ys).

/*   4.     no_doubles(L1, L2) where L2 is the result of removing all
duplicate elements from L1.
      ie: no_doubles([a,b,c,b], [a,c,b]) is true
      No_doubles accomplishes its task by breaking up the first list to
head and tail. The
      head is assigned to X and the tail to Xs, Ys is an unbound
variable. It then checks
      to see if the head element is a member of the rest of the list. If
it is, in which
      case, its a double, the rest of the list is sent back to the
relation an nothing is
      added to the unbound variable. The relation repeats, when the head
is not a member
      of the rest of the list, the head is double-checked to be a "non-
member" and that
      element is assigned to the head of the list Ys. This creates a new
list of
      "no_doubles. The relation ends with the base case of two empty
lists.*/

      no_doubles([], []).
      no_doubles([X|Xs], Ys):-
            member(X, Xs),
            no_doubles(Xs, Ys).
                  no_doubles([X|Xs], [X|Ys]):-
                  \+ member(X, Xs),
                        no_doubles(Xs, Ys).

      /*Member (X,Xs) is true if X is a member of the list Xs.*/
      member(X,[X|Xs]).
      member(X,[Y|Ys]):- member(X,Ys).


/*   5.Binary trees are represented by the functor tree
(Element,Left,Right), where Element is
     the element at the node, Left
     and Right are the left and right subtrees. The empty tree is
represented by the atom void.
     Using this notion, the tree:

                         a
                        / \
                       b   c

     is represented as:
     tree(a, tree(b, void, void), tree(c, void, void).
     Write a prolog program binary_tree(Tree) which is true if Tree is a
binary tree. */

/*
      binary_tree(Tree) :- Tree is a binary tree. This relation works by
checking each
      side of the tree to see if it has a left and a right child. The
relation stops when
      the value is void and no more children exitst.

      Sample call
      binary_tree(tree(a,tree(b,void,void),tree(c,void,void))).
      yes */

      binary_tree(void).
      binary_tree(tree(Element,Left,Right)) :-
            binary_tree(Left), binary_tree(Right).
```

```
/*Michael Amann
 CSE 240 Richard Whitehouse
 TTh @ 3:30-4:15
 Prolog Lab2
 Made with XEmacs!*/

/*Given the following relation templates for a coffee club factbase:
manager(Name), true if Name is a manager.*/

/*Each person in this factbase in a manager*/
      manager(galron).
      manager(bellana).
      manager(worf).
      manager(harry).

/*bill(Name, Number, Amount), true if Name has been sent a bill
numbered Number for Amount. paid(Number, Amount, Date), true if a
payment of Amount was made on Date for the bill
numbered Number for the amount.*/

/*Each person in the "bill" fact base has received a bill*/

      bill(galron,1,50).
      bill(bellana,2,75).
      bill(worf,3,105).
      bill(harry,4,8).
      bill(galron,5,100).
      bill(harry,6,200).
      bill(ensign,7,120).

/*Each person in the "paid" factbase paid their bill on the stated date
for the state amount*/

      paid(1,50,031601).
      paid(2,75,032101).
      paid(3,100,030201).

/*Define views of the factbase (rules and queries) to answer the
following questions: Which manager has been sent a bill for less than
ten dollars?*/

/*Less than ten returns the name of the manager that received a bill
for an amount less than ten dollars. First the person is verified to be
a manager, then the database searches for a bill in the amount of less
than ten dollars that is associated with the name of that manager.*/

less_than_ten(Name) :- manager(Name),
                   bill(Name,Number,Amount),
                        Amount @< 10.

/*Who has been sent more than one bill?
The relation "billed" searches the database for persons who received a
bill. It then looks again for another bill that has the same name as
the first. After comparing whether the number on the two bills matches,
if it doesn't, that person's name is returned.*/
```

```
billed(Name):- bill(Name,Number,Amount),
                  bill(Name1,Number1,Amount1),
                        Name == Name1,Number \== Number1.

/*Who has made a payment that is less than the amount of their bill?
The relation "past_due" checks the two facts bases, bill and paid. When
the number in bill matches the number in paid, the amounts are
compared. If the Amount in bill is larger than the Amount in paid, the
name is returned.*/

      past_due(Name):- bill(Name,Number,Amount),
                  paid(Number,Amount2,Date),
                        Number == Number,
                              Amount @> Amount2.

/*Who has received a bill and either not paid it at all, or not paid
their bill prior to a specified date? In the relation "not paid," the
two fact bases bill and paid are traversed. If there is a bill for a
person an no entry whatsoever in the paid database, the name is
returned.*/

      not_paid(Name):- bill(Name,Number,Amount),
                  \+paid(Number,Amount,Date).

/*In the relation "paid_late," the two factbases are searched. When the
Number in bill matches the Number in paid, the date is compared to the
one supplied by the user. If the date is less than or equal the date
supplied, the name is returned.*/

      paid_late(Name,Y):- bill(Name,Number,Amount),
                  paid(Number,Amount,Date),
                        Date @=< Y.

/*The relation "dead_beats," combines "paid_late" and "not_paid" to
search for entries in the database in which a person has not paid their
bill at all or has not paid by a specified date.*/

      dead_beats(Name,Y):- not_paid(Name).
                    :- paid_late(Name,Y).

/*Define rules for the following relations:
Write your own times(N1, N2, Prod) which is true if Prod is the product
of N1 and N2.In the "times" relation, the number provided by the user
are evaluated to see if they are products of each other. If so, yes is
returned. */

      times(N1,N2,Prod):- Prod is N1* N2.

      /*Sample call | ?- times(3,2,6).
                  yes */

/*Write your own prefix(Prefix, List) rule which is true if Prefix is a
prefix of List. In the "prefix" relation, append is used to return the
first element of a list and compare it to the element supplied by the
user to determine it is a prefix of a list.*/
```

```
      prefix(X,Y):- append([Y],_,X).

      /*Sample call: | ?- prefix([a,b,c], a).
                  yes */

/*Write your own suffix(Suffix, List) rule which is true if Suffix is a
suffix of List. In the "suffix" relation, append is used to return the
last element of a list and compare it to the element supplied by the
user to determine if it is a suffix of a list.*/

      suffix(X,Y):- append(_,[Y],X).

      /*Sample call: | ?- suffix([a,b,c], c).
                  yes */

/*Write your own sublist(Sub, List) rule which is true if Sub is
sublist of List. In the "sublist" relation member is used to parse
through the two supplied list and assign the first of each list to a
value Z. If Z in List1 ever matches Z in List2, yes is returned.*/

      member(X,[X|_]).
      member(X,[_|Tail]):- member(X,Tail).
      sub_list(X,Y):- member(Z,X),
                  member(Z,Y),
                        Z==Z.

      /*Sample calls? | ?- sub_list([a,b,c],[b]).

                  yes

                  | ?- sub_list([a,b,c],[d]).
                  no */

/*Write your own append(List1, List2, List3) rule which is true if
List3 is formed by joining List1 and List2 (List1List2).*/

      addhead(List,Element,[Element|List]).
            my_append(X,Y,N):- member(Z,X),
                  member(A,X), addhead(Y,Z,N),addhead(Y,A,N).

      /*Sample call ?- my_append([a],[b,c,d],N).
                  N = [a,b,c,d]     */

/*Using only the append relation, formulate rules to:
Determine the third element of a list. In this example we are not
concerned with the first two elements of the list. We simply append the
third element of the list with nothing and assign it to X.*/

      third(X,Y):- append([_,_,Y],_,X).

/*Determine the last element of a list. In this example, the last
element of the rest of the list is assigned to X and returned.*/

      last(X,Y):- append(_,[Y],X).
```

```
//*******************************************************************
//This program translates words and sentences into PigLatin.
//Michael A. Amann
//CSE 200 Richard Whitehouse, MWF @ 7:40
//*******************************************************************

import java.util.*;

public final class PigLatin {

/*Constructor with null body ensures objects of this class cannot be
instantiated.*/
    private PigLatin (){}

    /*Translate method takes string and tokenizes it.*/
    public static String translate (String inputString){

      String result = " ";

      /*StringTokenizer object is created to Tokenize string input.*/
      StringTokenizer tokenizer = new StringTokenizer (inputString);

      while(tokenizer.hasMoreTokens()) {

          result += translateWord(tokenizer.nextToken());
          result += " ";
      }
      /*The trailing space is removed from the tokens.*/
      result = result.trim();
      return result;

    }//method translate

    /*Translate word method translates string into PigLatin */
    private static String translateWord (String wordToBeTranslated){

      String firstpart;
      /*The resulting integer from the findFirstvowel method is
      assigned to position.*/
      int position = findFirstVowel(wordToBeTranslated);

      /*If a word ends in y, the y is removed and yay is appended.*/
      if(wordToBeTranslated.charAt((wordToBeTranslated.length()-1)) ==
      'y' && position == -1)
          {
            wordToBeTranslated = wordToBeTranslated.replace('y', ' ');
            wordToBeTranslated = wordToBeTranslated.trim();
            wordToBeTranslated = wordToBeTranslated + "yay";
          }

    else if(wordToBeTranslated.charAt((wordToBeTranslated.length()-
    1))== 'y'&& position == 0)
          {
            wordToBeTranslated = wordToBeTranslated.replace('y', ' ');
            wordToBeTranslated = wordToBeTranslated.trim();
            wordToBeTranslated = wordToBeTranslated + "yay";
          }
```

```
        /*If a word begins with a vowel, yay is appended to the word.*/
        else if(position == 0)

            wordToBeTranslated = wordToBeTranslated + "yay";

        /*If a word contains a vowel, the letters before the vowel are
        removed and appended to the end of the word, then ay is appended.
        */
        else if(position > 0)
            {
        wordToBeTranslated = wordToBeTranslated.substring(position)+
                  wordToBeTranslated.substring(0, position ) + "ay";
            }

        /*If a world contains no vowels, yay is appended to the word.*/
        else if(position == -1)

            wordToBeTranslated = wordToBeTranslated + "yay";

        return wordToBeTranslated;

      }//method translateword

     /*FindfirstVowel method checks for the occurence of a vowel in the
     string.*/
      private static int findFirstVowel (String word){

        boolean found = false;
        int index = 0;
        int firstvowel=0;

        while (!found && index < word.length()){

            if ("aeiouAEIOU".indexOf(word.charAt(index)) != -1)

              found = true;

            else
              index++;
        }
        if (found == false)
            return -1;
        else
            return index;

      }/*method find first vowel*/

      /*Main method calls the PigLatin class and submits a string for
      translation.*/
      public static void main (String[] args){

        System.out.println(PigLatin.translate("Romeo romeo where for art
        thou Romeo"));

      }/*method main*/

}/*class PigLatin*/
```