Data Structures — *CS-315 — Spring 20213*

Second Programming Project — Made available on Monday, February 13, 2023

Purpose

The purpose of this instructional unit and this programming assignment is to learn to think *recursively.*

Programming assignment

Consider the following abstract data type, which we shall call *recursive lists*, and which is defined recursively[1]!

1. The *empty list*, referred to as "null", and written as

   ```
   ( )
   ```

   is a recursive list. (The historical names for the empty list is nil.)

2. An *atom*, which is any string of letters (upper or lower case, with the two cases considered different) uninterrupted by digits, punctuation or blanks, is a recursive list. (For convenience we will limit atoms to strings of letters of length eight or less. More than eight characters will be treated as an error.) Typical atoms are

   ```
   a
   atom
   MARK
   Rivoire
   Melcon
   Neong
   Kooshesh
   ```

3. If l_1, l_2, l_3, $\ldots$, l_n are recursive lists, then so is

   $(l_1\ l_2\ l_3\quad l_n)$

   Typical (non-trivial) recursive lists are:

   ```
   (a b)
   (c)
   ( () a (b (c d)) ((e)) )
   ( ((a) b) (c b) )
   ```

   (The blanks have no significance, except when necessary to separate atoms.)

4. A recursive list is anything derivable from (1), (2) and (3) above and nothing else.

[1]To the best of my knowledge, this problem was initially proposed by Dr. Henry Shapiro at the University of New Mexico.

ADT Recursive list

The abstract data type *recursive list* has the following functions associated with it.

- `list null();`

  Return an empty list.

- `bool is_null(list p);`

  The function returns `true` (non-zero) if `p` is the empty list, otherwise it returns `false`. Therefore, the truth value of `is_null( null() )` is `true`.

- `bool is_atom(list p);`

  The function returns `true` if `p` is an atom, otherwise it returns `false`.

- `bool eq(list p, list q);`

  The function returns `true` if `p` and `q` are both atoms and are the same atom, otherwise it returns `false`, i.e., either `p` or `q` is not an atom or they are both atoms, but not the same atom.

- `list car(list p);`

  `p` is neither an atom nor the empty list. If `p` is an atom or the empty list, `car` is undefined (produces an error message and bombs out). Otherwise `car` returns the first element of `p`, i.e., returns l_1, which is necessarily a recursive list. For the recursive lists of (3) in the definition, `car` returns:

  ```
  a
  c
  ()
  ((a) b)
  ```

- `list cdr(list p);`

  `p` is neither an atom nor the empty list. If `p` is an atom or the empty list, `cdr` is undefined (produces an error message and bombs out). Otherwise `cdr` returns the recursive list formed by deleting the first element of `p`, i.e., returns (l_2 l_3 l_n), which is necessarily a recursive list. For the recursive lists of (3) in the definition, `cdr` returns:

  ```
  (b)
  ()
  ( a (b (c d)) ((e)) )
  ( (c b) )
  ```

- `list cons(list p, list q);`

  `q` is a non-atomic (a list that is not an atom) recursive list (may be the empty list). If `q` is an atom, `cons` is undefined (produces an error message and bombs out). Otherwise `cons` returns the recursive list formed by making `p` the first element of the result, with the original elements of `q` the second through $n+1$st elements of the result, i.e., returns (`p` $l_1 l_2 \dots l_n$), which is necessarily a recursive list.

  ```
  cons( (a (b)),((d) c) ) = ( (a (b)) (d) c)
  ```

  Note that `cons(car(p),cdr(p))` = `p`, assuming that `car(p)` and `cdr(p)` are defined.

- `void write_list(list p);`

  The function writes the human readable representation of a list to the standard output.

- `list read_list();`

  This function reads a recursive list written in human readable form (according to the rules described earlier) from the standard input and constructs an internal representation, which it returns.

The names `car`, `cdr` and `cons` are historical and are used in the Lisp programming language and its variants.

It is incredible how many interesting functions can be defined using:

- The `if` and `if-else` of C++.
- Calls to the basic routines described above.
- Recursive calls, to both the function being written and functions already written.

Note that the above list does *not* include looping constructs (`while`, `do-while`, and `for`) or even the use of variables, and the assignment operator (although you need to use the assignment operator to read a list from standard input).

A few examples

Consider the function `append`.

```
list append(list p, list q)
```

`p` and `q` are non-atomic lists (a list that is not an atom). The result is a single list that contains the elements of `p` followed by the elements of `q`. Some examples are:

```
append( ( ),( ) ) = ( )
append( (a b c),(c a) ) = (a b c c a)
append ( (a (b) (c d) ),( ((c)) ) ) = ( a (b) (c d) ((c)) )
```

Another function is `reverse`, which takes as an argument a list and returns a list with the elements in reverse order, i.e., ($l_n \quad l_3\ l_2\ l_1$). For the lists of (3) in the definition, reverse yields:

```
(b a)
(c)
( ((e)) (b (c d)) a () )
( (c b) ((a) b) )
```

In `main.cpp`, included in `http://www.cs.sonoma.edu/~cs315/reclists.tar`, you will find the definition of `append` and `reverse` and sample calls to these functions. You are being given the implementation for the list ADT; all you need to know is the type name `list` and the calling sequences for the standard functions. Also, recall that function arguments in C++ will not necessarily be evaluated from left to right. Thus, in the following call, the arguments to `append` may be passed in reverse order.

```
write_list( append( read_list(), read_list() ) )
```

See `main.cpp` to learn how to read and store lists before calling other functions.

Functions to write

Your assignment is to write, in the style of `append` and `reverse` (i.e., using recursive calls and `if` ... `else` ..., but no looping constructs and variables), an implementation for each of the following functions. In your solution to the following functions, you may only use the functions that are listed under *ADT Recursive list*, *append*, and *member* (shallow member).

Note that in the following description, the term "non-atomic list" means a list that is not an atom. That is, a list like `()` or `( a b c )`, or `( a () (b c) )`, but not like `abc`.

- `bool is_lat(list p);`

  `is_lat` takes a non-atomic (a list that is not an atom) list and returns `true` if the list is a list (potentially empty) of atoms. (It can bomb out if `p` is an atom, or you may check for errors and report them if you prefer.) For example

  ```
  is_lat( (a b c) ) = true
  is_lat( (a (b) c) ) = false
  ```

  Remember: No `while` loops.

- `bool member(list p, list q);`

  `p` is an atom and `q` is an non-atomic list. `member` returns `true` if `p` appears anywhere in `q`. (If `p` is not an atom or `q` is not a non-atomic list, the call is in error—you may detect this or just bomb out.)

- `list last(list p);`

  `last` returns the last element, l_n, of a non-atomic, non-empty list. (Do this without using `reverse`.) Note that last of `(a b c)` is `c`, not `(c)`.

- `list list_pair(list p,  list q);`

  `list_pair` takes two lists of atoms of the same length (you may check for an error if you like—or just bomb out) and returns a list that consists of lists of two atoms each, which are the corresponding atoms paired up. For example:

  ```
  list_pair( (a b c),(d e f) ) = ( (a d) (b e) (c f) )
  ```

- `list firsts(list p);`

  `firsts` takes as an argument a list whose elements are lists of atoms and returns a list which contains the first element from each of the lists. For example:

  ```
  firsts( ( (a b c) (d e f) (c d b a) ) ) = ( a d c )
  ```

- `list flat(list p);`

  `flat` takes a non-atomic list and returns a list which is the original list with the parenthesis removed (except for the outer set). For the lists in (3) of the definition, flat yields

  ```
  (a b)
  (c)
  (a b c d e)
  (a b c b)
  ```

- `bool two_the_same(list p, list q);`

  `two_the_same` takes two non-atomic recursive lists and returns `true` if `p` and `q` contain at least one atom in common.

- `bool equal(list p, list q);`

  `equal` takes two arbitrary recursive lists and determines if they are identical, that is the parentheses are all in the same place and the atoms agree as to place and name. This is an extension of `eq`.

- `list total_reverse(list p);`

  This function takes a recursive list and returns its mirror image. It is the extension of reverse that reverses the list and each sub-list, recursively, unto the nth generation.

- `list shape(list p);`

  `shape` takes a non-atomic recursive list and returns a recursive list that consists of only the parentheses in the original.

- `list intersection(list p,  list q);`

  `p` and `q` are lists of atoms with no atom appearing twice in `p` and no atom appearing twice in `q`. `p` and `q` represent sets. `intersection` forms their intersection. (The empty list represents the empty set.)

- `list list_union(list p, list q);`

  `list_union` forms the set which is their union—remember if `p` and `q` have an atom in common then this atom should only appear once in the union.

- `list permute(list p);`

  `p` is a list of atoms. `permute` returns a list whose elements are lists of atoms, each of which is a different permutation of the original list `p`. All the permutations of `p` should appear once and only once in what the function returns. For example:

  `permute( (a b c) )` = `( (a b c) (a c b) (b a c) (b c a) (c a b) (c b a) )`

  Unlike all the above problems the answer on this problem is not unique. The permutations (six in this example) can appear in any order.

This last problem is harder than the rest. You are not required to write this function. However, if you manage to write the permute, you get 10 points extra credit. You can not get any help for permute!

As mentioned above, you are going to be given the code for all supporting functions that you need to implement your code so you can develop your code on the platform of your choice[2].

Please note that *you need not to understand how these supporting functions are implemented.* You need to:

- clearly understand the primitive functions that have been discussed in this write-up so you can use them in your solutions to the required functions.
- add your solutions to each of the required functions to `solutions.cpp` and its function-prototype to `solutions.hpp`.
- use `main.cpp` to test your implementation.
- other than `solutions.hpp`, `solutions.cpp`, and `main.cpp`, do not under any circumstance modify the contents of any other file, including the `makefile`.

[2]However, your code should compile and run on our Linux server.

How we test your solution

We make a copy your `solutions.hpp` and `solutions.cpp` (notice the case of the letters in these two names and the fact that the header file uses hpp for extension) to a directory that contains a fresh copy of all supporting files, add our own `main.cpp` to it, compile and test your solution.

Input/Output

You do not write any function that requires input or output. These functions have been provided to you.

Grading

Your program will be graded for correctness (85%) and programming style (15%).