Symbolic Data in Scheme

1. Start DrRacket and type the following four lines exactly as shown into the top window, then click the *Run* button in the upper right corner of the window:

   ```
   (define a 'all)
   (define b 'these)
   (define c 'problems)
   (define d '())
   ```

   These lines define the symbols `a`, `b`, `c`, and `d` to be `all`, `these`, `problems`, and the *null list* `()`, respectively. To check if this worked, type the symbol `a` at the prompt by itself and hit Return. DrRacket should respond with the symbol `all`, indicating that the symbol `a` currently refers to the symbol `all`. Check the values of `b`, `c`, and `d` in the same way.

2. To build or extend a list, we can use `cons`. For example, evaluating `(cons c d)` creates the new list `(problems)`. Try this out, and then verify that `d` itself still refers to the null list. We can also nest `cons` expressions inside one another to build up bigger lists. For example, `(cons b (cons c d))` produces `(these problems)`. How would you produce `(all these problems)`? How about `(all these)`?

   Once you have an expression that builds the list `(all these)`, use it to help you build the slightly more complicated list `((all these) problems)`. Then create `cons` expressions to make the following lists:

   ```
   ((all) these problems)
   (all (these) problems)
   ((all these problems))
   (all (these problems))
   ```

3. Another way of creating lists is with the `list` function. `List` takes any number of values (possibly zero) and constructs a new list containing the values. For example, evaluating `(list a b c)` will produce the list `(all these problems)`. Evaluating `(list b)` will produce `(these)`. It's often easier to use `list` rather than `cons`, especially when building complicated lists. Try constructing the above four lists using `list` instead of `cons`.

4. If you evaluate the symbol `a` by itself at the Scheme prompt, you'll get the symbol `all`, because we defined `a` to be `all` in step 1 above. What happens if you evaluate the expression `'a` instead of `a` (that is, the symbol `a` preceded by a single apostrophe)? What happens if you evaluate `(list 'a 'b 'c)`? What about `(list a 'b c)`? Try these out and see what happens.

   In general, the single quote mark (apostrophe) tells Scheme to treat the expression that immediately follows it *literally*, rather than determining its *value*. You can also quote entire lists in the same way. For example, we could define the symbol `fruits` to be the list `(apples bananas cherries and peaches)` like so:

   ```
   (define fruits '(apples bananas cherries and peaches))
   ```

   Add this line to the top window, click *Run*, and then type `fruits` at the prompt to verify its value.

5. Now that we know how to build up lists, we need to know how to take them apart. For this we have the functions `car` and `cdr` (pronounced "could-er"). `Car` takes a list and returns its first element, without actually changing the list. For example, evaluating `(car fruits)` would give the symbol `apples`. Try this, then check `fruits` to make sure that it is still the same list as before.

   `Cdr` takes a list and gives back a new version of the list with the first element *removed*. Again, the original list itself is not changed. Try evaluating `(cdr fruits)` at the prompt. After doing this, make sure to check that `fruits` is still the same list as before. What does `(cdr (cdr fruits))` give? What about `(car (cdr (cdr fruits)))`? What about `(cons 'mangos (cdr fruits))`?

6. Now define the symbol `yum` to be the list `((spaghetti) and tofuballs)`. What is `(car yum)`? How can you retrieve just the symbol `spaghetti`, without the extra parentheses? (Hint: take the `car` of the `car`.) How can you retrieve just the symbol `and`? What about `tofuballs`?

7. What is
   (car (cdr (cdr (car *l*)))) where *l* is the list `((kiwis mangos lemons) and (more))`
   (car (cdr (cdr (car *l*)))) where *l* is the list `((() eggs and (bacon)) (for) (breakfast))`
   (car (cdr (cdr (cdr *l*)))) where *l* is the list `(() () () (and (coffee)) please)`

8. To get the symbol `and` in `(peanut butter and jelly on toast)` we can write (car (cdr (cdr *l*))). What would you write to get `Harry` in *l*,
   where *l* is the list `(apples in (Harry has a backyard))`
   where *l* is the list `(apples and Harry)`
   where *l* is the list `(((apples) and ((Harry))) in his backyard)`

9. What would you write to get the symbol `x` from *l*,
   where *l* is the list `(a (b c x) d)`
   where *l* is the list `(((x)))`
   where *l* is the list `((a (b)) (x) c)`
   where *l* is the list `(a (b (c (d x))))`

10. What do `(car ''apple)` and `(cdr ''apple)` evaluate to? Explain.

11. Add the following definitions to the top window exactly as shown (don't forget the ' marks) and click *Run*:

```
(define garfield 'cat)
(define cat 'garfield)
```

For each of the following boolean expressions, decide whether it will return true or false (`#t` or `#f`) before typing it into DrRacket. Then type it in and check your answer. Remember that the `eq?` function should only be used to test equality of symbols, not lists, whereas `equal?` works for any type of data.

a. `(equal? garfield 'cat)`

b. `(equal? 'garfield 'cat)`

c. `(eq? 'garfield 'cat)`

d. `(eq? garfield 'garfield)`

e. `(equal? (cons garfield '(cat)) '(garfield cat))`

f. `(equal? (cons 'garfield '(cat)) '(garfield cat))`

g. `(eq? (cons 'garfield '(cat)) '(garfield cat))`

h. `(equal? garfield garfield)`

i. `(equal? (car '(garfield cat)) garfield)`