CS61A Notes 15 – Logic Is What Logic Declares Logic To Be [Solutions v1.0]

Lists Again (and again, and again, and again, and again… Someone changed the Matrix!)

QUESTIONS

1. **Write a rule for `car` of list. For example, `(car (1 2 3 4) ?x)` would have `?x` bound to `1`.**
   ```
   (rule (car (?car . ?cdr) ?car))
   ```

2. **Write a rule for `cdr` of list. For example, `(cdr (1 2 3) ?y)` would have `?y` bound to `(2 3)`.**
   ```
   (rule (cdr (?car . ?cdr) ?cdr))
   ```

3. **Using the above, write a query that would bind `?x` to the `car` of `my-list`. Write another query that would bind `?y` to the `cdr` of `my-list`.**
   The temptation is to write (car my-list ?x) or (cdr my-list ?y). This doesn't work! There is no entry in the database whose first element is "car" and whose second element is the word "my-list". If you did that, you're thinking in the old Scheme way – that some "evaluator" will see my-list as a symbol and substitute in (1 2 3 4). This will not happen, since my-list isn't a variable! What you have to do is this:
   ```
   (and (my-list ?ls) (car ?ls ?x))
   ```
   First, we match ?ls to be (1 2 3 4), and then match ?x to be 1.

4. **Define our old friend, `member`, so that `(member 4 (1 2 3 4 5))` would be satisfied, and `(member 3 (4 5 6))` would not, and `(member 3 (1 2 (3 4) 5))` would not.**
   ```
   (rule (member ?item (?item . ?cdr)))
   (rule (member ?item (?car . ?cdr)) (member ?item ?cdr))
   ```

5. **Define its cousin, `deep-member`, so that `(deep-member 3 (1 2 (3 4) 5))` would be satisfied as well.**
   ```
   (rule (deep-member ?item (?item . ?cdr)))
   (rule (deep-member ?item (?car . ?cdr)) (deep-member ?item ?car))
   (rule (deep-member ?item (?car . ?cdr)) (deep-member ?item ?cdr))
   ```
   Note how ?item can either be in ?car or ?cdr, so we need three rules.

6. **Define another old friend, `reverse`, so that `(reverse (1 2 3) (3 2 1))` would be satisfied.**
   ```
   (rule (reverse () ()))
   (rule (reverse (?car . ?cdr) ?reversed-ls)
         (and (reverse ?cdr ?r-cdr)
              (append ?r-cdr (?car) ?reversed-ls)))
   ```

7. **(HARD!) Define its cousin, `deep-reverse`, so that**
   `(deep-reverse (1 2 (3 4) 5) (5 (4 3) 2 1))` would be satisfied.
   ```
   (rule (deep-reverse ?item ?item) (lisp-value atom? ?item))
   (rule (deep-reverse () ()))
   (rule (deep-reverse (?car . ?cdr) ?dr-ls)
         (and (deep-reverse ?car ?r-car)
              (deep-reverse ?cdr ?r-cdr)
              (append ?r-cdr (?r-car) ?dr-ls)))
   ```

We need the first rule because recall that a "deep-list" could be an atom, and that the third rule does not check if the ?car is an atom or not when it recurses on it.

8. **Write the rule `remove` so that `(remove 3 (1 2 3 4 3 2) ?what)` binds `?what` to `(1 2 4 2)` – the list with 3 removed.**

```
(rule (remove ?item () ()))
(rule (remove ?item (?item . ?cdr) ?result)
      (remove ?item ?cdr ?result))
(rule (remove ?item (?car . ?cdr) (?car . ?r-cdr))
      (and (not (same ?item ?car))
           (remove ?item ?cdr ?r-cdr)))
```

9. **Write the rule `interleave` so that `(interleave (1 2 3) (a b c d) ?what)` would bind `?what` to `(1 a 2 b 3 c d)`.**

```
(rule (interleave ?ls () ?ls))
(rule (interleave () ?ls ?ls))
(rule (interleave (?car . ?cdr) ?ls2 (?car . ?r-cdr))
      (interleave ?ls2 ?cdr ?r-cdr))
```

10. **Consider this, not very interesting rule: `!(listify ?x (?x))` . So if we do `(listify 3 ?what)`, `?what` would be bound to `(3)`.**

Define a rule map with syntax `(map` *procedure list result*`)`, so that `(map listify (1 2 3) ((1) (2) (3)))` would be satisfied, as would `(map reverse ((1 2) (3 4 5)) ((2 1) (5 4 3)))`. In fact, we should be able to do something cool like `(map ?what (1 2 3) ((1) (2) (3)))` and have `?what` bound to the word "`listify`". Assume the "procedure" we pass into `map` are of the form `(`*procedure-name argument result*`)`.

```
(rule (map ?proc () ()))
(rule (map ?proc (?car . ?cdr) (?new-car . ?new-cdr))
      (and (?proc ?car ?new-car)
           (map ?proc ?cdr ?new-cdr)))
```

11. **We can let predicates have the form `(`*predicate-name argument true/false*`)`. Define a rule `even` so that `(even 3)` is not be satisfied, and `(even 4)` is.**

```
(rule (even ?x) (lisp-value even? ?x))
```

12. **The above is a way to make predicates. And once we have predicates, we can – and will , of course – write a `filter` rule with the syntax `(filter` *predicate list result*`)` so that `(filter even (1 2 3 4 5 6) (2 4 6))` would be satisfied, and querying `(filter ?what (10 11 12 13) (10 12))` would bind `?what` to the word "`even`".**

```
(rule (filter ?pred () ()))
(rule (filter ?pred (?car . ?cdr) (?car . ?new-cdr))
      (and (?pred ?car)
           (filter ?pred ?cdr ?new-cdr)))
(rule (filter ?pred (?car . ?cdr) ?new-ls)
      (and (not (?pred ?car))
           (filter ?pred ?cdr ?new-ls)))
```