CS61A Notes – Week 15: Logic programming (solutions)

Lists Again (and again, and again, and again, and again…)

QUESTIONS

1. **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.

2. **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)))
   ```

3. **(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.

4. **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)))
   ```

5. **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))
   ```

6. **Consider this not very interesting rule: `(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 "procedures" 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)))
```

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

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

8. **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))` returns `Yes`, 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)))
```

Number Theory (The Bizarre Way)

QUESTIONS

1. **Write the rule `subtract` using the same syntax as `sum`. Assume that the first argument will always be greater than the second (since we don't support negative numbers with our system!)**

```
(rule (subtract ?x 0 ?x))       ;; x - 0 = x
(rule (subtract ?x (s ?y) ?z)   ;; x - y = z <=> x - (y-1) = z + 1
      (subtract ?x ?y (s ?z)))
```

2. **Write the rule `product`.**

```
(rule (product 0 ?x 0))        ;; 0 * x = 0
(rule (product (s ?x) ?y ?z)   ;; x * y = z <=> (x-1) * y + y = z
      (and (product ?x ?y ?i)
           (sum ?i ?y ?z))))
```

3. **Define the rule `exp` (for exponent, of course), with the first argument the base and second the power, so that `(exp (s (s 0)) (s (s 0)) ?what)` would bind `?what` to `(s (s (s (s 0))))`.**

```
(rule (exp ?x 0 (s 0)))         ;; x^0 = 1
(rule (exp ?base (s ?pow) ?z)   ;; x^y = z <=> x^(y-1) * x = z
      (and (exp ?base ?pow ?i)
           (product ?base ?i ?z)))
```

4. **Write the rule `factorial`, so that `(factorial (s (s (s 0))) ?what)` would bind `?what` to `(s (s (s (s (s (s 0))))))`.**

```
(rule (factorial 0 (s 0)))    ;; 0! = 1
(rule (factorial (s ?x) ?y)   ;; x! = y <=> (x-1)! * x = y
      (and (factorial ?x ?i)
           (product ?i (s ?x) ?y)))
```

5. **Write the rule `max` with syntax `(max number1 number2 the-bigger-number)` so that `(max (s 0) (s (s 0)) ?what)` would bind `?what` to `(s (s 0))`.**

```
(rule (max ?x 0 ?x))
(rule (max 0 ?x ?x))
(rule (max (s ?x) (s ?y) (s ?z))
      (max ?x ?y ?z))
```

6. **Write the rule `appearances` that counts how many times something appears in a list. For example, `(appearances 3 (1 2 3 3 2 3 3) ?what)` would bind `?what` to `(s (s (s (s 0))))`.**

```
(rule (appearances ?item () 0))
(rule (appearances ?item (?item . ?cdr) (s ?count))
      (appearances ?item ?cdr ?count))
(rule (appearances ?item (?car . ?cdr) ?count)
      (and (not (same ?car ?item))
           (appearances ?item ?cdr ?count)))
```

7. **Note that we can represent negative numbers by putting `s` to the right. For example, negative two would be `((0 s) s)`. Write `*-1` so that `(*-1 ((0 s) s) ?what)` binds `?what` to `(s (s 0))` and `(*-1 (s (s 0)) ?what)` binds `?what` to `((0 s) s)`.**

```
(rule (*-1 0 0))           ;; -1 * 0 = 0
(rule (*-1 (s ?x) (?y s)) ;; -1 * x = y <=> -1 * (x-1) = y + 1
      (*-1 ?x ?y))
(rule (*-1 (?x s) (s ?y)) ;; -1 * x = y <=> -1 * (x+1) = y - 1
      (*-1 ?x ?y))
```

Note there are three cases: the first argument is 0, is a positive number, or is a negative number.

8. **(HARD!) Write rules `sum2` and `subtract2` that can take in negative numbers.**

Note that there are five general cases for sum2: one of the numbers is 0, the first number is positive and the second negative, the first number is negative and the second positive, both positive, and both negative.

The case with 0 is trivial:

```
(rule (sum2 0 ?x ?x))
(rule (sum2 ?x 0 ?x))
```

In the case of both positive, we can just use the sum that we wrote before.

```
(rule (sum2 (s ?x) (s ?y) ?z)
      (sum (s ?x) (s ?y) ?z)) ; use old sum
```

If first is negative and second is positive, note that

```
-x + y = z <==> (-x+1) + (y-1) = z
```

```
(rule (sum2 (?x s) (s ?y) ?z)
      (sum2 ?x ?y ?z))
```

If the first is positive and the second is negative, we take advantage of addition's commutative property, and use the above.

```
x + -y = z <==> -y + x = z
```

```
(rule (sum2 (s ?x) (?y s) ?z)
      (sum2 (?y s) (s ?x) ?z)) ;; sum is commutative!
```

If both are negative, note that:

```
-x + -y = -1 * (x + y)
```

```
(rule (sum2 (?x s) (?y s) ?z)
      (and (*-1 (?x s) ?pos-x)
           (*-1 (?y s) ?pos-y)
           (sum2 ?pos-x ?pos-y ?pos-z)
           (*-1 ?z ?pos-z)))
```

Since our sum2 can deal with negative numbers now, note that

```
x – y = x + -y
```

```
(rule (subtract2 ?x ?y ?z)
      (and (*-1 ?y ?-1*y)
           (sum2 ?x ?-1*y ?z)))
```