CS61A Notes – Week 07b: Nondeterministic and Logic programming (solutions)

Nondeterministic and Indecisive

QUESTIONS

1. **Suppose we type the following into the amb evaluator:**
```
> (* 2 (if (amb #t #f #t)
           (amb 3 4)
           5))
```
What are all possible answers we can get?

```
6, 8, 10, 6, 8
```

2. **Write a function `an-atom-of` that dispenses the atomic elements of a deep list (not including empty lists). For example,**
```
> (an-atom-of '((a) ((b (c))))) => a
> try-again => b
```

```
(define (an-atom-of ls)
    (cond ((null? ls) (amb))
          ((atom? ls) ls)
          (else (amb (an-atom-of (car ls))
                     (an-atom-of (cdr ls))))))
```

3. **Use `an-atom-of` to write `deep-member?`.**

```
(define (deep-member? X ls)
    (let ((maybe-x (an-atom-of ls)))
        (require (equal? x maybe-x))
        #t))
```

4. **Fill in the blanks:**
```
> (define (choose-member L R)
     (cond ((null? R) (amb))
           ((= (car L) (car R)) (car L))
           (else (amb (choose-member L (cdr R))
                      (choose-member (cdr L) R)))))
> (choose-member '(1 2 3) '(4 2 3))
3

> try-again
2

> try-again
2
```

Lists Again (and again, and again, and again, and again...)

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. 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))
```

4. 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.

5. 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)))
```

6. (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.

7. 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)))
```

8. 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))
```

9. 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)))
```

10. 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))
```

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

```
Alternatively,
```

```
(rule (subtract ?a ?b ?c)        ;; a - b = c <=> c + b = a
      (sum ?c ?b ?a))
```

2. Write the rule `product`. You may use rules that you have defined before.

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

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

3. Write the rule `divide` that divides the first argument with the second, and returns the quotient and the remainder. For example, `(divide (a a a a a a a) (a a a) ?quo ?rem)` would bind `?quo` to `(a a)` and `?rem` to `(a)`. You may (and should!) use rules that you have defined before.

```
Recall that (quotient * divisor) + remainder = dividend.
```

```
(rule (divide ?dividend ?divisor ?quo ?rem)
      (and (product ?quo ?divisor ?prod)
           (sum ?prod ?rem ?dividend)))
```

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

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

5. Write the rule `factorial`, so that `(factorial (a a a) ?what)` would bind `?what` to `(a a a a a a)`.

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

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 `(a a a a)`.

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