CS61A Notes – Week 14 – Logic Programming

This week we introduced Logic (declarative) programming in lecture, which embodies a very different way of thinking than the more traditional imperative programming you've been using so far in the course. A declarative "program" is a description of the desired characteristics of a solution. The system determines these characteristics through a process called unification. Instead of defining functions to operate on data, the programmer describes relationships between variables. We have our own declarative language that uses scheme like syntax and is available on the instructional machines by typing *scmlog* into the interpreter.

scmlog is a program that interprets scheme expressions as logical assertions. We can define facts that have a conclusion and zero or more hypothesis. Once some facts have been established, we can have the system solve for logical variables, which are denoted by an underscore.

(fact Conclusion Hypothesis1 Hypothesis2 . . .)
(? (Conclusion))

Here is an example of declarative program written in our scheme-prolog syntax. Try to guess what the rules describe!

```
(fact (parent george paul))
(fact (parent martin george))
(fact (parent martin martin_jr))
(fact (parent martin donald))
(fact (parent donald ann))

(fact (parent _X _Y) (mother _X _Y))
(fact (parent _X _Y) (father _X _Y))

(fact (ancestor _X _X))
(fact (ancestor _X _Y) (parent _X _Z) (ancestor _Z _Y))
```

```
scmlog > (? (parent _X george))
_X : martin
```

1. What does scmlog return for the following queries (Yes is printed if the expression can be satisfied, No is printed otherwise)?

a. (? (parent _x ann))
donald

b. (? (ancestor ann martin))
No.

b. (? (ancestor martin paul))
Yes.

2. Now that you understand the basic syntax of scheme-prolog and have seen some examples, let's try something a bit more interesting. Fill in the cousin fact that determines whether two people are cousins. It should accept three symbols: Two people and a True/False symbol depending on if the they are cousins.

(fact (cousin _X _Y) (parent _A _X) (parent _B _A) (parent _Z _Y) (parent _B _Z))

Unfortunately for the programmer (you), scmlog doesn't have a built in number system! Fortunately, us Berkeley students are savvy individuals and can come up with our own. Here is a description of our simple but elegant number system: We've defined it to start with Zero, and we increment using the idea of a successor. We'll call one the successor of zero, two the successor of one, and so on. For example, 5 would be represented as (Suc . (Suc . (Suc . (Suc . (Suc . Zero))))). The dot notation allows us to deal with pairs.

3. Given the description of our number system, try to write a predicate even in which the second symbol is True if the first symbol is an even "number," and False if it's odd.

We've started even for you, and here's an example of how it would be run:

```
scmlog > (? (even (Suc . (Suc . Zero)) _P))

_P : True
```

```
(fact (even Zero True))
(fact (even (Suc . _X) True) (even _X False))
(fact (even (Suc . _X) False) (even _X True))
```

4. Just testing for even numbers is a bit boring. What's a number system without additions? Implement the add fact below (the base case is written for your convenience). Convince yourself that multiplication and exponentiation would be just as easy.

```
(fact (add Zero _Y _Y))
(fact (add (Suc . _X) _Y (Suc . _Z)) (add _X _Y _Z))
```

5. Now let's combine the two examples we've seen so far. Describe a method of counting the distance between two people. We'll say the distance between a person and him/her self is Zero. Use the number system from above, and take a look at ancestor if you're confused.

```
(fact (ancestor-dist _X _X Zero))
(fact (ancestor-dist _X _Y (Successor . _N)) (parent _X _Z) (ancestor-dist _Z _Y _N) )
```

6. Finally, let's write some list operations. We'll start with writing append, where the first two arguments are lists and the third is the result of appending them. The dot notation will come

in handy here. Writing (_first _rest) allows us to break up a list into its first and rest, just like an rlist.

a.

```
(fact (append () _B _B))
(fact (append _B _B))
(fact (append (_X . _rest) _B (_X . _Z)) (append _rest _B _Z))
```

c. Time for some recursion! Try writing eqlist, which tests two lists to see if they're equal.

```
(fact (eqlist () ()))
(fact (eqlist (_X . _R1) (_X . _R2)) (eqlist _R1 _R2))
```

d. Write reverse, who's first symbol (“argument”) is a list and the second is the reversed version.

```
(fact (reverse () ()))
(fact (reverse (_X . _L) _R) (reverse _L _R1) (append _R1 (_X) _R))
```

e. Now write a fact that checks if a list is a palindrome. Hint – you've already written reverse and eqlist...

```
(fact (palindrome _X) (reverse _X _Z) (eqlist _X _Z))
```

f. Finally, let's try translating the higher order function filter to scheme-prolog. We've defined a simple predicate that tests for the number 3 (and only works on numbers between 1 and 4.

```
(fact (is3 1 False))
(fact (is3 2 False))
(fact (is3 3 True))
(fact (is3 4 False))

(? (filter is3 (1 3 3 2 2 4 3) _X))
_X : (1 2 2 4)

(fact (filter _P () ()))
(fact (filter _P (_X . _R) (_X . _S)) (filter _P _R _S) (_P _X True))
(fact (filter _P (_X . _R) _S) (filter _P _R _S) (_P _X False))
```