LOGIC 12

COMPUTER SCIENCE 61A

August 2, 2016

1 Introduction

Over the semester, we have been using *imperative programming* – a programming style where code is written as a set of instructions for the computer. In this section, we introduce *declarative programming* – code that declares *what* we want, not *how* to do it. Logic programming (what we are learning) is a type of declarative programming.

In this class, we will be using a language called Logic. The Logic language borrows syntax from Scheme and ideas from Prolog.

2 Simple Facts and Queries

In Logic, you can define *facts*. Facts are simply Scheme lists of relations and relations are simply Scheme lists of symbols. Remember, relations are **not** call expressions; instead, relations are used to express relationships between symbols.

Here's an example of a fact:

```
> (fact (sells supermarket groceries))
```

This line of code says: "This is a fact: supermarkets sell groceries". When we declare something as a fact, we are simply saying that it is a **true statement**. You can think of a fact as an axiom, i.e., something that is fundamentally true.

"sells" is a quality that relates two things, "supermarket" and "groceries." What are the values of "supermarket" and "groceries"? They have no values! They are *symbols* – symbols are Logic's primitives.

Having defined some facts, we can make *queries* – in other words, we can ask Logic for information based on the facts that we've defined:

```
> (query (sells supermarket groceries))
Success!
> (query (sells supermarket books))
Failed.
> (query (sells supermarket ?stuff))
Success!
stuff: groceries
```

The first query asks, "Is it a fact that supermarkets sell groceries?" and the second query asks, "Is it a fact that supermarkets sell books?". The third query above is equivalent to asking "What do supermarkets sell?" Logic replies that supermarkets sell groceries, *based on the previously defined fact*.

Note that `?stuff` is a variable in Logic, whereas `supermarket` is a symbol. `supermarket` is always going to be `supermarket`, but `?stuff` is unknown – it is only *after* the query that we know what the value of `?stuff` is.

We can also query both multiple elements of a relation:

```
> (query (sells ?place ?stuff))
Success!
place: supermarket stuff: groceries
```

This is equivalent to asking "What are places that sell stuff, and what stuff do they sell?" Logic will tell you what each variable should be based on previously defined facts.

2.1 Simple Questions

1. Write a fact that checks if two elements are equal.

2. Define a set of facts about complementary nucleotides. Remember from biology that
   - **A**denine and **T**hymine are complementary to each other
   - **C**ytosine and **G**uanine are complementary to each other

3 Complex Facts

In Logic, you can also define more complex facts. For example:

```
> (fact (sells-same ?store1 ?store2)
        (sells ?store1 ?item)
        (sells ?store2 ?item))
```

Here is the basic syntax of a complex fact:

```
> (fact (<conclusion>)
        (<hypothesis 1>)
        (<hypothesis 2>)
        ...
        (<hypothesis n>))
```

This is equivalent to saying "the conclusion is true if all the hypotheses are true." If even one of the hypotheses is false, the conclusion cannot be proven using this fact.

For example, the `sells-same` complex fact is equivalent to saying "`store1` and `store2` sell the same thing if `store1` sells `item` and `store2` also sells the same `item`."

You can perform fact-checking with complex facts, just like with simple facts:

```
> (fact (sells farmers-market groceries))
> (fact (sells starbucks coffee))
> (query (sells-same supermarket farmers-market))
Success!
> (query (sells-same supermarket starbucks))
Failed.
```

We can also do querying:

```
> (query (sells-same ?store supermarket))
Success!
store: farmers-market
```

This is equivalent to asking "what store sells the same thing as a supermarket?"

We can also ask "what stores sell the same thing?"

```
> (query (sells-same ?store1 ?store2))
Success!
store1: supermarket store2: farmers-market
store1: farmers-market store2: supermarket
store1: supermarket store2: supermarket
store1: farmers-market store2: farmers-market
```

3.1 Questions

1. Write simple and complex facts for `in`, a relation between a symbol and a list that is satisfied if and only if that symbol is in the list.

```
> (query (in a (a b c)))
Success!
> (query (in d (a b c)))
Failed.
> (query (in ?elem (a b c)))
Success!
elem: a
elem: b
elem: c
```

2. Write simple and complex facts for `every-other`, a relation between two lists that is satisfied if and only if the second list is the same as the first list, but with every other element removed.

```
> (query (every-other (frodo merry sam pippin) ?x))
Success!
x: (frodo sam)
> (query (every-other (gandalf) ?x))
Success!
x: (gandalf)
```

3. Write simple and complex facts for `mapped`, a relation between a relation and two lists that is satisfied if and only if each element of the second list satisfies the relation with the corresponding element of the first list. (The example uses the `complementary` fact you defined at the beginning of discussion.)

```
> (query (mapped complementary (g t a g t a g t a) ?nyan))
Success!
nyan: (c a t c a t c a t)
```

4. Write facts for `prefix`, a relation between two lists that is satisfied if and only if elements of the first list are the first elements of the second list, in order.

```
> (query (prefix (being for the) (being for the
                  benefit of mister kite)))
Success!
> (query (prefix (for no one) (for no one)))
Success!
> (query (prefix () (got to get you into my life)))
Success!
> (query (prefix (want i to) (i want to hold your hand)))
Failed.
```

5. Write facts for `sublist`, a relation between two lists that is satisfied if and only if the first is a consecutive sublist of the second. For example:

```
> (query (sublist (give) (never gonna give you up)))
Success!
> (query (sublist (you up) (never gonna give you up)))
Success!
> (query (sublist () (never gonna give you up)))
Success!
> (query (sublist (never give up) (never gonna give you up)))
Failed.
> (query (sublist (let you down) (never gonna give you up)))
Failed.
```

   Hint: You will want to use the prefix fact that you previously defined.

6. Write facts to implement the `subs` relation that relates two symbols `old` and `new` with two lists `input` and `output`. The relation holds whenever `output` is identical to `input`, except with every occurrence of `old` replaced by `new`.

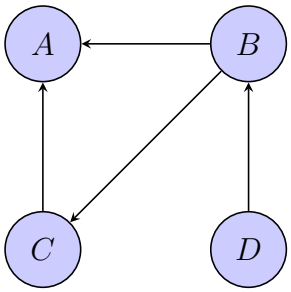
   Hint: You may need the `equal` relation from earlier.

```
> (query (subs romeo fred (romeo oh romeo wherefore art thou
   romeo) ?x))
Success!
x: (fred oh fred wherefore art thou fred)
```

3.2 Extra Questions

Similarly to the map coloring example from lecture, we can construct the graph shown on the right in Logic using some simple facts:

```
> (fact (vertices A B C D))
> (fact (edge B A))
> (fact (edge B C))
> (fact (edge C A))
> (fact (edge D B))
```



We can implement some pretty interesting relations for this graph. In these questions we will implement the Hamiltonian path relation, which is true for a vertex if there a path starting from that vertex that visits each node once. For example, in the given graph, `(D B C A)` is a Hamiltonian path, and this is actually the only one. Let's start with a few helper relations.

1. Implement facts for the `remove` relation, which relates `elem` and two lists if, and only if, the second list has all and only the elements of the first list but with one occurrence of `elem` removed. `elem` must appear at least once in the first list.

```
> (query (remove a (b a n a n a s) ?lst))
Success!
lst: (b n a n a s)
lst: (b a n n a s)
lst: (b a n a n s)
> (query (remove no (not in this list) ?lst))
Failed.
```

2. Use `remove` to implement fact for the `contain-same` relation, which relates two lists that contain exactly the same elements, though not necessarily in the same order.

```
> (query (contain-same (A B C D) (D C A B)))
Success!
> (query (contain-same (A B C) (B B C A)))
Failed.
```

3. Our last helper relation is aptly named `hamil-helper`, a relation between a vertex `v` and a list of vertices `so-far` that is satisfied if the rest of a Hamiltonian path can be constructed from `v` given that we have already visited the vertices in `so-far`. `contain-same` may be useful here.

```
> (query (hamil-helper A (B C D)))
Success!
> (query (hamil-helper A (B C)))
Failed.
```

4. Finally, write `hamiltonian`, a relation that holds for a vertex `v` if there is a Hamiltonian path starting from `v`. This should be very easy now!

```
> (query (hamiltonian D))
Success!
> (query (hamiltonian B))
Failed.
```