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 was based on the Scheme project and also borrows a few 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:

```lisp
> (query (sells supermarket groceries))
Success!
> (query (sells supermarket books))
Failure.
> (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.

A similar query is

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

This query is equivalent to asking “Which places sell groceries?” Once again, Logic replies based on the previously defined fact.

We can also query both parameters:

```lisp
> (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.

In Logic, we can also model hierarchical data by nesting relations inside of other relations. For example:

```lisp
(fact (person (name bob) (age 49)))
(fact (person (name alice) (age 20)))
```

declares two facts. The first fact asserts that there exists a person whose name is Bob and whose age is 49. The second fact asserts that there exists a person whose name is Alice and whose age is 20.
Moreover, we can query the facts that we previously defined:

```lisp
> (query (person (name ?first_name) (age 49)))
Success!
first_name: bob
> (query (person (name bob) ?age))
Success!
age: (age 20)
```

The first query asks, “What is the name of a person whose age is 49?” and the second query asks, “What is the age of a person named Bob?”.

### 2.1 Questions

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

2. Define a set of facts for a “mall,” which has the following qualities:
   - malls sell shoes and clothes
   - malls are larger than supermarkets
   - malls are popular

3. Define a set of facts to model the table of data below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Color</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulbasaur</td>
<td>001</td>
<td>Green</td>
<td>Grass</td>
</tr>
<tr>
<td>Charmander</td>
<td>004</td>
<td>Red</td>
<td>Fire</td>
</tr>
<tr>
<td>Squirtle</td>
<td>007</td>
<td>Blue</td>
<td>Water</td>
</tr>
<tr>
<td>Caterpie</td>
<td>010</td>
<td>Green</td>
<td>Bug</td>
</tr>
<tr>
<td>Pikachu</td>
<td>025</td>
<td>Yellow</td>
<td>Electric</td>
</tr>
</tbody>
</table>
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")
     ("hypothesis1")
     ("hypothesis2")
     etc.
)
```

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 will also be false.

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))
Failure.
```

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
```
3.1 Questions

1. 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.

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

2. 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.

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

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

4. Write a set of facts to implement the subs relation with components old, new, input, and output. The first two are symbols; the last two can be symbols or lists. The output should be the same as the input except that every appearance of old is replaced by new.

> (query (subs romeo fred (romeo oh romeo why art thou romeo) ?x))
Success!

x: (fred oh fred why art thou fred)