LOGIC PROGRAMMING AND UNIFICATION 12

COMPUTER SCIENCE 61A

April 25, 2013

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:

```
> (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.

A similar query is

```
> (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 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.

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

```
(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:

```
> (query (person (name ?first-name) (age 49)))
Success!
first-name: bob
> (query (person (name bob) ?age))
Success!
age: (age 49)
```

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.

   Solution:

   ```
   (fact (equal ?x ?x))
   ```

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

   Solution:

   ```
   (fact (sells mall shoes))
   (fact (sells mall clothes))
   (fact (larger mall supermarkets))
   (fact (popular mall))
   ```

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

Name	Number	Color	Type
Bulbasaur	001	Green	Grass
Charmander	004	Red	Fire
Squirtle	007	Blue	Water
Caterpie	010	Green	Bug
Pikachu	025	Yellow	Electric

Solution:

```
> (fact (pokemon (name bulbasaur) (number 1) (color green)
  (type grass)))
> (fact (pokemon (name charmander) (number 4) (color red)
  (type fire)))
> (fact (pokemon (name squirtle) (number 7) (color blue)
  (type water)))
> (fact (pokemon (name caterpie) (number 10) (color green)
  (type bug)))
> (fact (pokemon (name pikachu) (number 25) (color yellow)
  (type electric)))
```

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'')
      (``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 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
```

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.

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

   Solution:

   ```
   (fact (every-other () ()))
   (fact (every-other ?x ?x))
   (fact (every-other (?a ?b . ?l-rest) (?a . ?r-rest))
         (every-other ?l-rest ?r-rest)
   )
   ```

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.

   ```
   > (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.
```

Solution:

```
(fact (prefix () ?any-list))
(fact (prefix (?first . ?small) (?first . ?big))
      (prefix ?small ?big)
)
```

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)))
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.

Solution:

```
(fact (sublist ?a ?b) (prefix ?a ?b))

(fact (sublist ?sub (?first . ?rest)) (sublist ?sub ?rest))
```

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

Solution:

```
(fact (subs ?old ?new () () )
)

(fact (subs ?old ?new (?old . ?rest1) (?new . ?rest2))
      (subs ?old ?new ?rest1 ?rest2)
)

(fact (subs ?old ?new (?x . ?rest1) (?x . ?rest2))
      (subs ?old ?new ?rest1 ?rest2)
)

(fact (subs ?old ?new (?first1 . ?rest1) (?first2 . ?rest2))
      (subs ?old ?new ?first1 ?first2)
      (subs ?old ?new ?rest1 ?rest2)
)
```

Note: this solution does not work quite as intended because of this case:

```
(fact (subs ?old ?new (?x . ?rest1) (?x . ?rest2))
      (subs ?old ?new ?rest1 ?rest2)
)
```

We have no way of specifying that the value represented by the variable ?x is not the same value represented by ?old or ?new. Therefore, we get much more output than the above "doctest" query implies (try it out to see!).

4 Unification

One of the basic operations that we try to compute in our logic interpreter is to *unify* two relations. In its simplest form, unification is finding an assignment for variables that makes two relations the same.

Recall from lecture that unification recursively unifies each pair of corresponding elements in two relations, accumulating an assignment. First, we must look up variables in the current environment. Then, we must establish new bindings to unify those elements.

Let's take a look at an example and walk through the execution process.

```
(?first ?first ?second ?second)

((a b ?x) (a ?y c) (a c ?y) (a ?x b))
```

First, we lookup the first value, `?first` which can be substituted with `(a b ?x)`. Then, we lookup the next variable, `?first` which is the same as the initial variable that we looked up. We assign `?first` the value of `(a ?y c)`. Since the variables were the same, we must verify that the two values are equivalent. So, we check to see if there is valid matching between `(a b ?x)` and `(a ?y c)`. We verify that is true by letting `?x` be `c` and `?y` be `b`. We repeat this process for the rest of the items in the relation.

Now that we've had some practice, let's work on a few problems.

4.1 Questions

1. Which of the following relations can be unified?

   (a) `(?x ?y ?x)`

   (b) `(a (a b) a)`

   (c) `(?x ?z a)`

   (d) `(?z (?x ?y) ?x)`

   Solution:

   ```
   (a) and (b), {x: a, y: (a b)}
   (a) and (c), {x: a, y: z}
   (b) and (c), {x: a, z: (a b)}
   (b) and (d), {x: a, y: b, z: a}
   ```

2. Which of the following relations can be unified?

   (a) `(?x (1 2) (3 ?y))`

   (b) `(?y (1 ?y) (?x ?x))`

   (c) `(?z ?y ?z)`

   (d) `((1 2) ?x ?x)`

Solution:

```
(a) and (c), {x: (3 (1 2)), y: (1 2), z: (3 (1 2))}
(c) and (d), {x: (1 2), y: (1 2), z: (1 2)}
```

Let's walk through the algorithm for unification. The unify function is implemented via a recursive process which keeps performing unification on corresponding parts of two expressions until a contradiction is reached or a viable binding to all variables can be established.

Let us begin with an example. The pattern (?x ?x) can match the pattern ((a ?y c) (a b ?z)) because there is an expression with no variables that matches both: ((a b c) (a b c)). Unification identifies this solution via the following steps:

1. To match the first element of each pattern, the variable ?x is bound to the expression (a ?y c).
2. To match the second element of each pattern, first the variable ?x is replaced by its value. Then, (a ?y c) is matched to (a b ?z) by binding ?y to b and ?z to c.

As a result, the bindings placed in the environment passed to unify contain entries for ?x, ?y, and ?z:

```
>>> e = read_line("(?x ?x)")
>>> f = read_line(" ((a ?y c) (a b ?z))")
>>> env = Frame(None)
>>> unify(e, f, env)
True
>>> env.bindings
{'?z': 'c', '?y': 'b', '?x': Pair('a', Pair('?y', Pair('c', nil)))}
```

The result of unification may bind a variable to an expression that also contains variables, as we see above with ?x bound to (a ?y c). The bind function recursively and repeatedly binds all variables to their values in an expression until no bound variables remain.

```
>>> print(bind(e, env))
((a b c) (a b c))
```

In general, unification proceeds by checking several conditions. The implementation of unify directly follows the description below.

1. Both inputs e and f are replaced by their values if they are variables.
2. If e and f are equal, unification succeeds.
3. If e is a variable, unification succeeds and e is bound to f.
4. If f is a variable, unification succeeds and f is bound to e.
5. If neither is a variable, both are not lists, and they are not equal, then e and f cannot be unified, and so unification fails.

6. If none of these cases holds, then e and f are both pairs, and so unification is performed on both their first and second corresponding elements.

Here is the actual code for reference:

```
def unify(e, f, env):
      """Destructively extend ENV so as to unify (make equal)
      e and f, returning True if this succeeds and False
      otherwise.  ENV may be modified in either case
      (its existing bindings are never changed)."""
      e = lookup(e, env)
      f = lookup(f, env)
      if e == f:
          return True
      elif isvar(e):
          env.define(e, f)
          return True
      elif isvar(f):
          env.define(f, e)
          return True
      elif scheme_atomp(e) or scheme_atomp(f):
          return False
      else:
          return unify(e.first, f.first, env) and \
                 unify(e.second, f.second, env)
```