LOGICAL PROGRAMMING 13

COMPUTER SCIENCE 61AS

The Basics of Logic Programming and How it Works

1. What are the two central operations that the query system is organized around?

   Solution: Pattern Matching and Unification

2. How do each of these work?

   Solution: Pattern Matching - A pattern matcher is a program that tests whether some datum fits a specified pattern. It takes a pattern, a datum, and a frame that specifies bindings for various pattern variables as inputs. If the datum matches the pattern and is consistent with the bindings already in the frame, it returns the frame.

   Unification - Unification allows both the "pattern" and "datum" to contain variables. The unifier takes two patterns and determines if it is possible to assign values to the variables that will make the two patterns equal.

3. How is pattern testing against frames organized and how does it work?

   Solution: It is organized through the use of streams. The system takes an input streams of frames. Given a single frame, the matching process runs through the database entries one by one and indicates whether that frame failed or is an extension to the frame.

4. How do compound queries work with pattern testing? (Describe for both `and` and `or`)

   Solution: For `and`, the process for the first query is run, then the second query uses the output stream of the first query as its input. For `or`, the two queries are run separately then merged.

5. What will the logic interpreter print if a rule returns true?

Solution: the logic interpreter will return the input.

Practice with the logic interpreter

1. Write logic (query) language rules for prefix, a relation between two lists that is satisfied if and only if the elements of the first list are the first elements of the second list, in order. Do not use lisp-value! For example, each of these examples matches the relation:

```
(prefix (being for the) (being for the benefit of mister kite))
(prefix (for no one) (for no one))
(prefix () (got to get you into my life))
```

But these do not satisfy the relation:

```
(prefix (want i to) (i want to hold your hand))
(prefix (to hold your) (i want to hold your hand))
(prefix (i want to tell you) (i want to))
```

Solution: The third example gives you a big hint about the base case, which is that the empty list is a prefix of any list.

```
(assert! (rule (prefix () ?any)))
```

If the first list isn't empty, the first element must be the first element of the second list and prefix must work on the cdrs of the two lists.

```
(assert! (rule (prefix (?car. ?fragment) (?car . ?original))
                       (prefix (?fragment ?original))))
```

2. Now write rules for sublist, a relation between two lists that is satisfied if and only if the first is a consecutive sublist of the second. Do not use lisp-value! Hint: You will want to use the prefix relation from part (a) to help you. For example, each of these examples matches the relation:

```
(sublist (give) (never gonna give you up))
(sublist (you up) (never gonna give you up))
(sublist () (never gonna give you up))
```

And these do not:

```
(sublist (never give up) (never gonna give you up))
(sublist (let you down) (never gonna give you up))
```

Solution: You should use PREFIX to help define `sublist`. This gives you the base case:

```
(assert! (rule (sublist ?a ?b)
           (prefix ?a ?b)))
```

If the first list isn't a prefix of the second list, then it has to be a sublist of the cdr of the second list:

```
(assert! (rule (sublist ?sub (?first . ?rest))
           (sublist ?sub ?rest)))
```