CS61A Week 15 | Logic Is What Logic Declares Logic To Be (v1.0)

Paradigm Shift Again

With this many paradigm shifts in a single semester, we expect you to at least be able to pronounce the word "paradigm" correctly after this class.

To reiterate, we are now in the realm of *logic* or *declarative* programming, at least for a week. Here in the magical world of non-imperative, we can say exactly what we want - and have the computer figure out how to get it for us. Instead of saying *how* to get the solution, we describe - declare - *what* the solution looks like.

The mind-boggling part of all of this is that it all just works through pattern matching. That is, there are no "procedures" in the way you're used to; when you write out a parenthesized statement, it's not really a procedure call, and you don't really get a return value. Instead, either you get entries from some database, or nothing at all.

Be Assertive And Ask Questions

There are two things you can type into our query system: an **assertion** and a **query**.

A **query** asks whether a given expression matches some fact that's already in the database. If the query matches, then the system prints out all matches in the database. If the query doesn't match to anything, you get no results.

An **assertion** states something true; it adds an entry to the database. You can either assert a simple fact, or a class of facts (specified by a "rule).

So here's an assertion that you've seen: `(assert! (wei likes chicken-feet))`

You can also assert rules. In general, a rule looks like: `(rule <conclusion> <subconditions>)`

And it's read: "conclusion is true if and only if all the subconditions are true". Note that you don't have to have subconditions! Here's a very simple rule:

```
(rule (same ?x ?x))
```

The above says two things satisfy the "same" relation if they can be bound to the same variable. It's deceptively simple - no subconditions are provided to check the "sameness" of the two arguments. Instead, either the query system can bind the two arguments to the same variable ?x - and it can only do so if the two are equivalent - or, the query system can't.

And the rule of love:

```
(assert! (rule (?person1 loves ?person2)
                        (and (?person1 likes ?something)
                             (?person2 likes ?something)
                             (not (same ?person1 ?person2))))))
```

The above rule means that `?person1` loves `?person2` if the three conditions following can all be satisfied - that is, `?person1` likes `?something` that `?person2` also likes, and that `?person1` is not the same person as `?person2`.

Note the `and` special form enclosing the three conditions - an entry in the database must satisfy ALL three conditions to be a match for the query. If you would like a rule to be satisfied by this or that condition, you can either use the

or special form in the obvious way, or you can make two separate rules. For example, if one loves another if they like the same things or if `?person1` is a parent of `?person2`, we would add the following to the database:

```
(assert! (rule (?person1 loves ?person2) (parent ?person1 ?person2)))
```

Note the new rule does NOT overwrite the previous rule; this is not the same thing as redefining a procedure in Scheme. Instead, the new rule complements the previous rule.

To add to confusion, you can also use the and special form for queries. For example,

```
(and (wei loves ?someone) (?someone likes chicken-feet))
```

is a query that finds a person Wei loves because that person likes chicken-feet.

There's another special form called lisp-value: `(lisp-value <pred?> <arg1, arg2, ...>)`

The lisp-value condition is satisfied if the given `pred?` applied to the list of args return #t. For example, `(list-value even? 4)` is satisfied, but `(lisp-value < 3 4 1)` is not. lisp-value is useful mostly for numerical comparisons (things that the logic system isn't so great at).

A note on writing rules: it's often tempting to think in terms of procedures - "this rule takes in so and so, and returns such and such". This is not the right way to approach these problems - remember, nothing is returned; an expression or query either has a match, or it doesn't. So often, you need to have both the "arguments" and the "return value" in the expression of the rule, and the rule is satisfied if "return value" is what would be returned if the rule were a normal Scheme procedure given the "arguments". Always keep in mind that everything is a Yes or No question, and your rule can only say if something is a correct answer or not. So when you write a rule, instead of trying to "build" to a solution like you've been doing in Scheme, think of it as trying to check if a given solution is correct or not.

In fact, this is so important I'll say it again: when you define rules, don't think of it as defining procedures in the traditional sense. Instead, think of it as, given arguments and a proposed answer, check if the answer is correct. The proposed answer can either be derived from the arguments, or it can't.

A different approach for writing declarative rules is to try to convert a Scheme program to a rule. For example, let's take a crack at the popular append:

```
(define (append ls1 ls2)
    (cond ((null? ls1) ls2)
          (else (cons (car ls1) (append (cdr ls1) ls2)))))
```

The `cond` specifies an "either-or" relationship; either `ls1` is null or it is not. This implies that, for logic programming, we'd need two separate rules, each corresponding to each `cond` clause. The first one is straightforward:

```
(rule (append () ?ls2 ?ls2))
```

The second `cond` clause breaks `ls1` into two parts - its car and its cdr - and basically says that appending `ls1` and `ls2` is the same as consing the first element of `ls1` to the list obtained by appending the cdr of `ls1` to `ls2`. Translated to logic-programming, it means the cdr of the result is equivalent to appending the cdr of `ls1` to `ls2`, and that the car of the result is just the car of `ls1`. This implies:

```
(rule (append (?car . ?cdr) ?ls2 (?car . ?r-cdr))
    (append ?cdr ?ls2 ?r-cdr))
```

We will try the above techniques on some of the harder problems below.

QUESTIONS:

1. Write a rule for car of list. For example, `(car (1 2 3 4) ?x)` would have `?x` bound to 1.

2. Write a rule for cdr of list. For example, `(cdr (1 2 3) ?y)` would have ?y bound to `(2 3)`.

3. Using the above, write a query that would bind `?x` to the car of `my-list`. Write another query that would bind `?y` to the cdr of `my-list`.

4. Define our old friend, `member`, so that `(member 4 (1 2 3 4 5))` would be satisfied, and `(member 3 (4 5 6))` would not, and `(member 3 (1 2 (3 4) 5))` would not.

5. Define its cousin, `deep-member`, so that `(deep-member 3 (1 2 (3 4) 5))` would be satisfied as well.

6. Write the rule interleave so that `(interleave (1 2 3) (a b c d) ?what)` would bind `?what` to `(1 a 2 b 3 c d)`.

Final Cramming CheckList

Week 1: Basic Scheme / Recursion

Week 2: Applicative vs. Normal / Lambda / Higher Order Functions

Week 3: Recursive vs. Iterative / Order of Growth

Week 4: Invariants / Debugging / Commenting

Week 5: Pair and Lists / Data Abstraction

Week 6: Trees / Deep Recursion / Scheme-1

Week 7: Data-Directed Programming / Message Passing / Tables

Week 8: Object Oriented Programming

Week 9: Environment Diagrams / OOP in vanilla scheme

Week 10: List Mutations / Vector / Scheme-2

Week 11: Client and Server/ Concurrency

Week 12: Meta-Circular Evaluator / Analyzing Evaluator / Logo

Week 13: Streams

Week 14: Lazy Evaluator / Non-deterministic Evaluator

Week 15: Logic Programming