Lecture 37: Declarative Programming

Announcements:

- Autograder should start running this weekend.
- Remember: you still have to provide your own tests!
- We have updated files other than scheme.py a few times. I generally announce if the modification is important. You can use Unix `diff` to check for differences between what you have and the version in `~cs61a/lib/projects/scheme`.
- Submit your Project 4 contest entries as "`proj4-contest`." by next Wednesday. Assuming we get entries, we'll ask the class to judge these entries.

Imperative vs. Declarative

- So far, our programs are explicit directions for solving a problem; the problem itself is *implicit* in the program.
- *Declarative* programming turns this around:
  - A "program" is a description of the desired characteristics of a solution.
  - It is up to the system to figure out how to achieve these characteristics.
- Taken to the extreme, this is a very difficult problem in AI.
- However, people have come up with interesting compromises for small problems.
- For example, *constraint solvers* allow you to specify relationships between objects (like minimum or maximum distances) and then try to find configurations of those objects that meet the constraints.

Prolog and Predecessors

- Way back in 1959, researchers at CMU created GPS (General Problem Solver [A. Newell, J. C. Shaw, H. A. Simon])
  - Input defined objects and allowable operations on them, plus a description of the desired outcome.
  - Output consisted of a sequence of operations to bring the outcome about.
  - Only worked for small problems, unsurprisingly.
- *Planner* at MIT [C. Hewitt, 1969] was another programming language for theorem proving: one specified desired goal assertion, and system would find rules to apply to demonstrate the assertion. Again, this didn't scale all that well.
- Planner was one inspiration for the development of the *logic-programming language Prolog*.

Prolog (Lisp Style)

- Let's interpret Scheme expressions as *logical assertions*.
- For example, `(likes brian potstickers)` might be such an assertion: `likes` is a *predicate* that relates `brian` and `potstickers`.
- We don't interpret the arguments of the predicate: they are just uninterpreted data structures.
- We also allow one other type of expression: a symbol that starts with an underscore will indicate a *logical variable*.
- An assertion such as `(likes brian _X)` asserts that there is some replacement for `_X` that makes the assertion true.

Facts and Rules

- We will make *queries* in the form of assertions, possibly with logical variables.
- The system will look to see if the queries are true based on a database of facts (axioms or postulates) about the predicates.
- It will inform us of what replacements for logical variables make the assertion true.
- Each fact will have the form

  `(fact` *Conclusion Hypothesis1 Hypothesis2* ...`)`

  Meaning "For any substitution of logical variables in the Conclusion and Hypotheses, we may derive the conclusion if we can derive each of the hypotheses."

Example: Family Relations

First, some facts with no hypotheses:

```
(fact (parent george paul))
(fact (parent martin george))
(fact (parent martin martin_jr))
(fact (parent martin donald))
(fact (parent george ann))
```

Now some general rules about relations:

```
(fact (ancestor _X _Y) (parent _X _Y))
(fact (ancestor _X _Y) (parent _X _Z) (ancestor _Z _Y))
```

From these, we ought to be able to conclude that Martin is an ancestor of Ann, for example.

Relations, Not Functions

- In this style of programming, we don't define functions, but rather relations.
- Instead of saying `abs(-3) == 3`, we say `abs(-3, 3)` (that is, "3 stands in the `abs` relation to -3."
- Instead of `add(x, y) == z`, we say `add(x, y, z)`.
- This will allow us to run programs "both ways": from inputs to outputs, or from outputs to inputs.