Lecture 31: Declarative Programming
**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.
Structured Query Language (SQL)

- For example, the database world has *relational databases* and *object-relational databases*, which represent relations between data values as *tables*, such as:

<table>
<thead>
<tr>
<th>ID</th>
<th>Last Name</th>
<th>First Names</th>
<th>Level</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>29313921</td>
<td>Smith</td>
<td>Michelle</td>
<td>2</td>
<td>3.6</td>
</tr>
<tr>
<td>38474822</td>
<td>Jones</td>
<td>Scott</td>
<td>3</td>
<td>3.2</td>
</tr>
<tr>
<td>89472648</td>
<td>Chan</td>
<td>John</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>48837284</td>
<td>Thompson</td>
<td>Carol</td>
<td>3</td>
<td>3.7</td>
</tr>
</tbody>
</table>

- SQL is a language for making *queries* against these tables:

```sql
SELECT * FROM Students WHERE level='2';
```

which selects the first and third *rows* of this table.

- We don’t say *how* to find these rows, just the criteria they must satisfy.

- So SQL can be thought of as a kind of declarative programming language.
Prolog and Predecessors

• Way back in 1959, researchers at Carnegie-Mellon University 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)

- In our sample language, the data values are (uninterpreted) Scheme values.
- Some of these values will be deemed to be “true.”
- A logic program tells us which ones.
- As for Scheme, we’ll write logic programs using Scheme data; you tell the data from the program by how it is used.
- For example, \texttt{(likes brian potstickers)} might be such an assertion:

  \begin{quote}
  \texttt{likes} is a \textit{predicate} that relates \texttt{brian} and \texttt{potstickers}.
  \end{quote}

- We don’t interpret the arguments of the predicate: they are just uninterpreted data structures.
Logical Variables

- We also allow one other type of expression: a symbol that starts with '?' will indicate a logical variable.
- Logical variables can stand for any possible Scheme value (including one that contains logical variables).
- As an assertion, `(likes brian ?X)` says that any replacement of ?X that makes the assertion true.
- As a query, `(likes brian ?X)` asks if there exists any value for ?X that makes the query true.
- When the same logical variable occurs multiple times in an expression, it is replaced uniformly.
- For example, `(<= ?X ?X)` might assert that everything is less than or equal to itself (or ask if there is anything less than or equal to itself).
Facts and Rules

• 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, we enter some facts with no hypotheses (with our logic system prompt to emphasize that this is not regular Scheme):

  ```scheme
  logic> (fact (parent george paul))
  logic> (fact (parent martin george))
  logic> (fact (parent martin martin_jr))
  logic> (fact (parent martin donald))
  logic> (fact (parent martin robert))
  logic> (fact (parent george ann))
  ```

- We can now ask specific questions, such as

  ```scheme
  logic> (query (parent martin george))
  Success!
  ```
Existential Queries

- With logical variables, we can find everything that satisfies a relation.

```logic
> (query (parent martin ?who))
Success!
who: george
who: martin_jr
who: donald
who: robert
```
Multiple Criteria

• We also allow queries in which multiple criteria must be satisfied:

\[
\text{logic>} \ (\text{query} \ (\text{parent} \ ?gp \ ?p) \ (\text{parent} \ ?p \ ?c))
\]
Success!

\[
\begin{array}{c}
gp: \ martin \\
p: \ george \\
c: \ paul \\
\end{array}
\]

\[
\begin{array}{c}
gp: \ martin \\
p: \ george \\
c: \ ann \\
\end{array}
\]

• As illustrated here, \( ?p \) is always replaced with the same value in both clauses in which it appears.
The Closed World

logic> (fact (parent george paul))
logic> (fact (parent martin george))
logic> (fact (parent martin martin_jr))
logic> (fact (parent martin donald))
logic> (fact (parent george ann))

logic> (query (parent martin paul))
Failed.

- Here, the facts don’t imply that Martin is the parent of Paul, so the query fails.
- Of course, in real life it does not follow that just because you don’t know something, it’s false.
- However, our system makes the “closed world assumption”: Anything not derivable from the given facts is false.
- On the other hand, the system is not set up to draw conclusions from this...
- ...so can’t define (non-ancestor ?x ?y) to be true if one can’t prove (ancestor ?x ?y).
Compound Facts

- Now some general rules about relations:

  \[
  \text{logic}> \quad \text{(fact (grandparent ?X ?Y) (parent ?X ?Z) (parent ?Z ?Y))}
  \]

- The general form is

  \[(\text{Conclusion} \ \text{Hypothesis1} \ \text{Hypothesis2}...\)]

- From these, we ought to be able to conclude that Martin is Ann’s grandparent, for example.
Recursive Facts

• Now let's generalize grandparent to ancestor:

logic> (fact (ancestor ?X ?Y) (parent ?X ?Y))

• That is, an ancestor is either your parent, or a parent of one of your ancestors (recursively).
Relations, Not Functions

• In this style of programming, we don’t define functions, but rather relations.
  
  - Instead of saying “(abs -3) yields 3”,...
  - We say “(abs -3 3) is true” (or, “-3 stands in the abs relation to 3.”)
  - Instead of “(add x y) yields z”,...
  - We say “(add x y z) is true.”

• The distinction between operand and result is eliminated.

• This will allow us to run programs “both ways”: from inputs to outputs, or from outputs to inputs.