

## Lecture 30: Declarative Programming

## Declarative vs. Imperative

- Up until now, we have dealt with what is called *imperative programming*.
- That is, our programs consist of commands that tell *how to* perform some computation.
- An alternative is *declarative programming*.
- Here, our programs *describe* the desired characteristics of the result, and leave it to the system to figure out how to get them.
- If feasible, it would seem that declarative programming is preferable, but that it is, of course, difficult to achieve.
- One current version is *programming by example*, in which one gives examples of desired results and tries to get the system to generalize this into an algorithm.
- Unsurprisingly, this hasn't quite made it to the main stream.
- There is a history of systems for making logical assertions about a problem domain and having the system figure out how to solve a problem in that domain.

## Prolog and Predecessors

- Way back in 1959, AI 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 the desired goal assertion to be proven, 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 was originally developed for AI applications (esp. natural-language processing) and is still in use for a variety of applications in AI.
- Figured in the Fifth Generation Computer Systems Project in Japan in the 1980s.

## 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: as far as Scheme is concerned they are just uninterpreted data structures.
- We also allow one other type of expression: a symbol that starts with a question mark 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))`
- Intended meanings: May deduce that george is paul's parent, etc.
- Now some general rules about relations:  
`(fact (ancestor ?X ?Y) (parent ?X ?Y))`  
`(fact (ancestor ?X ?Y) (parent ?X ?Z) (ancestor ?Z ?Y))`
- Intended meanings:
  - For any values of `?X` and `?Y`, if we can deduce that `?X` is `?Y`'s parent, then we may deduce that `?X` is `?Y`'s ancestor.
  - For any values of `?X`, `?Y`, and `?Z`, if we can deduce that `?X` is `?Z`'s parent, and `?Z` is `?Y`'s ancestor, then we may deduce that `?X` is `?Y`'s ancestor.

## Example, continued

```
(fact (parent george paul))      (fact (parent martin george))
(fact (parent martin martin_jr)) (fact (parent martin donald))
(fact (parent george ann))
(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, or find out someone's ancestors or descendants.

```
logic> (query (ancestor martin ann))
Success!
logic> (query (ancestor ?who ann))
Success!
who: george
who: martin
logic> (query (ancestor martin ?who))
Success!
who: george
who: martin_jr
who: donald
who: paul
who: ann
```

## 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.

## Recap: A “Schemish” Prolog

- As a query, a Scheme expression, e.g. `(ordered (0 1 2))` represents a logical assertion.
  - Its top-level operator (e.g., `ordered`) names a *predicate* (true/false function).
  - Its operands are the data for this predicate: unlike Scheme programs, they don't represent function calls—they are the literal data...
  - ... with the exception that *logical variables*, represented as symbols starting with '?', stand for operands that may be replaced by other expressions.
- To define a predicate, we give rules for it:

`(fact CONCLUSION)` means that `CONCLUSION` is to be taken as true, for any replacement of its logical variables.

`(fact CONCLUSION HYPOTHESIS ...)` means that `CONCLUSION` is to be taken as true, assuming that the `HYPOTHESES` can all be shown to be true. Again, this is for all replacements of logical variables throughout the rule.

## Operational and Declarative Meanings

- Thus,  
`(fact (eats ?P ?F) (hungry ?P) (has ?P ?F) (likes ?P ?F))`  
means that for any replacement of `?P` (e.g., 'brian') and `?F` (e.g., 'potstickers') throughout the rule:
  - Declarative Meaning** If brian is hungry and has potstickers and likes potstickers, then brian will eat potstickers.
  - Operational Meaning** To show that brian will eat potstickers, show that brian is hungry, then that brian has potstickers, and then that brian likes potstickers.
- The *declarative meaning* allows us to look at our Scheme-Prolog program as a *logical specification* of a problem for which the system is to find a solution.
- The *operational meaning* allows us to look at our Scheme-Prolog specification as an *executable program* for searching for a solution.
- **Closed Universe Assumption:** We make only positive statements. The closest we come to saying that something is false is to say that we can't prove it.

## Demo

(Text of demos available in the files accompanying this lecture.)

## How It's Done (I): Unification

- In general, our system, given a target expression involving a predicate to prove, must find a fact that might assert that target, given a suitable replacement of logical variables.
- To do this, we try to pattern-match the conclusions of all our facts against the target expression.
- The pattern matching is called *unification*, [J. A. Robinson].
- For example, we say that `(likes brian potstickers)` *unifies with* the expression `(likes ?P ?F)`, if we substitute `brian` for `?P` and `potstickers` for `?F`.
- Might think of this substitution—called a *unifier*—as a Python dictionary mapping logical variables to expressions.
- Just for completeness, let's take a look at one possible approach to creating these unifiers.

## Unification (II)

- The substitution has to be uniform:
  - Can unify `(le 0 1)` with `(le ?X ?Y)`
  - But cannot unify `(le 0 1)` with `(le ?X ?X)`
- Everything is symmetric: if  $A$  unifies with  $B$ , then  $B$  unifies with  $A$ . Logical variables can appear in one or both.
- It is possible for logical variables to be unified with each other:  
Unify `(likes ?P ?F)` with `(likes ?Q potstickers)`.
- We substitute `potstickers` for `?F`, and choose either to substitute `?Q` for `?P` or vice-versa.
- The result in either case means that any person likes potstickers.

## Implementing Logical Variables and Substitutions

- A logical variable (`?x`) may be bound to any Scheme expression, including a logical variable.
- The set of all these bindings is called a *unifier*.
- Unifiers are like environments, but work a little differently.
- If `?x` is bound to `?y`, then `?x` is also bound to anything `?y` is bound to.
- At that point, binding *either* `?x` or `?y` to something other than a logical variable binds *both* of them to that thing.
- Initially, every logical variable is bound to itself.

## Implementing Logical Variables and Substitutions (II)

- Main operations on unifiers are `bind` and `binding`:

```
class Unifier:
    def __init__(self, init={}):
        self.bindings = dict(init) # Makes a copy

    def binding(self, expr):
        """Current binding of EXPR. If EXPR is not a logical
        variable, always returns EXPR itself."""
        while expr in self.bindings:
            expr = self.bindings[expr]
        return expr

    def bind(self, var, value):
        assert is_logical_var(var)
        self.bindings[var] = value
```

- Can use ability to copy environments to *back out* of an attempted match.

## Implementing Unification

A simple tree recursion with side-effects:

```
def unify(E0, E1, unif):
    """Returns True iff E0 and E1 can be unified by an extension
    of UNIF. UNIF is modified to be this extension."""
    def unify1(E0, E1):
        E0 = unif.binding(E0); E1 = unif.binding(E1)
        if scheme_eqp(E0, E1): return True
        if is_logical_var(E0):
            unif.bind(E0, E1) # E0 is always unbound here
            return True
        elif is_logical_var(E1):
            unif.bind(E1, E0) # E1 is always unbound here
            return True
        elif scheme_atomp(E0) or scheme_atomp(E1): return False
        else:
            return unify1(E0.first, E1.first) \
                and unify1(E0.rest, E1.rest)

    return unify1(E0, E1)
```

## Other Examples

- In the next lectures, we'll look at a couple of useful, but highly specialized declarative tools for programs that involve string manipulation.
- *Regular expressions* are patterns that describe strings. The system provides the means to match these patterns to strings.
- *Backus-Naur Form (BNF)* is another kind of pattern-matching facility that is more powerful than regular expressions. It is useful for describing things like programming languages in a human-readable form that also lends itself to actual language implementation.