Week 12
Prolog overview

A language designed for A.I.

Logics programming paradigm

- Programmer specifies relationships among possible data values.
- User poses queries.
  
  *What data value(s) will make this predicate true?*

Comparison with functional paradigm (Clojure, etc.)

- A functional program consists of a sequence of function definitions.
- A logic program consists of a sequence of relation definitions.
- "The Prolog execution environment doesn't so much compute an answer, it deduces an answer from the relation definitions at hand. Rather than being given an expression to evaluate, the Prolog environment is given an expression which it interprets as a question."
  
  -- Bucknell University Prolog Manual.

Prolog is different from other programming languages

- Prolog has no types. In fact, the basic logic programming environment has no literal values as such.
- Identifiers starting with lower-case letters denote data values, while all other identifiers denote variables.
- Though the basic elements of Prolog are typeless, most implementations have been enhanced to include character and integer values and operations.
- Also, Prolog has mechanisms for describing tuples and lists.
- See
  
Underlying concept of Prolog

- Built upon symbolic **predicate logic**.
  - Closer to pure mathematics than most programming languages.
  - Small and simple

- Statements specify boolean values:
  - facts (true)
  - rules (predicates)

But Prolog does not have
- assignments
- branches
- loops
- conditional statements

**But how can we express a process without those?**

Status in 2016

- Prolog (~1972) is a much newer language than Lisp (1958).
- Nevertheless, Prolog standardization was late.
  - Early examples use some different symbols.
  - We've settled on **ISO Prolog** (1995) as a standard.

  - Read it and try to understand everything
  - But skip sections that don't seem to apply to our examples.

Asserting a fact

- A simple property is a 1-argument function:
  - dog(snoopy). *Snoopy is a dog*
  - prime(13). *13 is a prime number*

- A binary relation is a 2-argument function:
  a. A subject-object or other non-symmetric relation
     - likes(mary,chocolate). *Mary likes chocolate*
     - mother(elizabeth,charles). *Elizabeth is the mother of Charles*
     - larger(Chicago, Baltimore)
  b. Or sometimes a symmetric relation
     - brother(william,harry). *Harry and William are brothers*

Are those all the facts?

- No. Factual predicates can have any number of parameters:

- But 1- and 2-argument predicates are often the most useful in A.I.
  - More complicated relationships can often be represented by multiple facts.
Expressing a rule

- A simple rule is an implication
  - like the predicate logic $P \rightarrow Q$

- Except that the conclusion is on the left
  - $Q \leftarrow P$

- Notation:
  - Some older Prologs used $\leftarrow$
  - Some simulated it by $<=$
  - Most (including the standard) now use $:-$

Backward chaining

- A query asks:
  
  "For what parameter values [variables in the expression] does the expression evaluate to true?"

Logical connectives

- **Conjunction** (logical and) is expressed by comma (or in some versions &)
  
  ```prolog
daughter(X,Y) :- child(X,Y), female(Y).
  \(Y\) is the daughter of \(X\)"

- **Disjunction** (inclusive OR) is expressed by multiple definition clauses:
  
  ```prolog
  parent(X,Y) :- daughter(Y,X).
  parent(X,Y) :- son(Y,X).
  \(X\) is a parent of \(Y\) assuming son and daughter already defined as primitive"

An important convention

- **Variables** (predicate parameters) begin with an upper-case letter.
  - Many Prolog experts like to use single-character names for variables (as in elementary algebra).
  - That's OK for short scopes, but in longer, more complicated code we apply similar criteria to choosing names in any programming language.

- Variables are bound in each statement.
  - So if \(Q\) appears three times in the same statement Prolog tries to instantiate all three occurrences to the same value.
  - But if \(Q\) also appears in the next statement, it's a different variable.
No circular definitions

Example: Three ways of defining parent relations:

a. Taking mother and father as primitive:
   parent(X,Y) :- mother(X,Y).
   parent(X,Y) :- father(X,Y).

b. Taking parent as primitive:
   mother(X,Y) :- parent(X,Y), female(X).
   father(X,Y) :- parent(X,Y), male(X).

c. Taking child as primitive (inverse relation)
   parent(X,Y) :- child(Y,X).

Which is best?

Properties of binary relations
(terminology review)

In elementary predicate logic we learned that a binary relation \( R \), using Prolog notation, is:

- Reflexive if \( R(X,X) \) is true for every \( X \) in the domain
- Symmetric if \( R(Y,X) \) is true whenever \( R(X,Y) \) is true.
- Transitive if \( R(X,Z) \) is true whenever \( R(X,Y) \) and \( R(Y,Z) \) are both true.
- Asymmetric if \( R(Y,X) \) is false whenever \( R(X,Y) \) is true.

What about: Antisymmetric? Irreflexive?

Which are most useful in A.I. (inference)?

Some kinds of relations

- An equivalence relation (also called an RST relation) is reflexive, symmetric, and transitive -- example: equals(X,Y)
- A simple ordering relation is transitive and antisymmetric -- example: subset(B,A)
- A strict simple ordering relation is transitive and irreflexive -- example: olderThan(X,Y)
- In designing rules of inference we can and should take advantage of those to avoid many special cases.

Choices

In the simple example (family relations) we chose which relations to take as primitive (data base) and which to define in terms of the others (rule base).

To keep it simple we prefer to work bottom-up starting with the most primitive relations that we're going to assert as facts.

- mother(elizabeth,charles)
- male(charles).
- or possibly the inverse
- son(charles,elizabeth).
- female(elizabeth).

What other choices were there?
Generally

- The simplest predicates should be the ones that we plan to assert as facts.
  
  ```prolog
  son(charles,elizabeth).
  ```

- We can then define more complicated and more general rules:
  
  ```prolog
  child(X,Y)  :- son(X,Y).
  child(X,Y)  :- daughter(X,Y).
  mother(X,Y) :- child(Y,X), female(X).
  ```

  *Keep it simple! Avoid redundancy!*

An important distinction

- Note that we cannot define *circular* relationships, even though some dictionaries may do so:
  
  ```prolog
  parent(X,Y):- child(Y,X).
  child(X,Y) :- son(X,Y).
  child(X,Y) :- daughter(X,Y).
  son(X,Y)   :- parent(Y,X), male(Y).
  ```

- In developing a very large rule base, we may forget and do that by accident.
  
  - Most Prolog processors will tell us where the problem is.
  
  - But we can and should . . .

Exploit recursion where it makes sense

- Especially in defining a transitive relation:
  
  ```prolog
  ancestor(X,Y):= parent(X,Y).
  ancestor(X,Y):= parent(X,W),ancestor(W,Y).
  ```

- Note the implied existential qualifier:
  
  - there exists a $W$ such that . . .
  
  - Prolog will try to instantiate it.

  ```prolog
  How will prolog evaluate
  ancestor(george5,william).
  What intermediate values will it instantiate?
  ```

But be careful!

- What's wrong with this?
  
  ```prolog
  sibling(X,Y):= parent(W,X), parent(W,Y).
  ```

  - i.e. X and Y are siblings if they have a common parent. Isn't that reasonable?

- How can we fix it?
  
  (Oops! We need to make it *irreflexive*. That's easy!)

  ```prolog
  sibling(X,Y) := X\=Y, parent(W,X), parent(W,Y).
  ```
Recursive query example

- ancestor(george5, william).
- parent(george5, edward8)
  & ancestor(edward8, william).

Failed! Try another choice.
Backtrack to try another person whose parent was george5

Recursive query example

- ancestor(george5, william).
- parent(george5, george6),
  ancestor(george6, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  ancestor(elizabeth, william).

Recursive query example

- ancestor(george5, william).
- parent(george5, george6),
  ancestor(george6, william).
- parent(george5, george6),
  ancestor(george6, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  ancestor(elizabeth, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  parent(elizabeth, charles),
  ancestor(charles, william).
Recursive query example

- ancestor(george5, william).
- parent(george5, george6),
  ancestor(george6, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  ancestor(elizabeth, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  parent(elizabeth, charles),
  ancestor(charles, william).
- parent(george5, george6),
  parent(george6, elizabeth),
  parent(elizabeth, charles),
  parent(charles, william).

Success!

Three kinds of 2-place query

- ancestor(george5, charles).
  Is it true that...?

- ancestor(george5, X).
  Who are George5's descendants?
  i.e. Whom is George5 the ancestor of?

- ancestor(X, George5).
  Who are George5's ancestors?
  i.e. Whom is George5 the descendant of?

Stopping recursion

- As in procedural programming, the base (non-recursive) cases should usually be the first ones specified. This (below) could lead to an infinite recursion.

  ancestor(X,Y) := parent(X,W), ancestor(W,Y).
  ancestor(X,Y) := parent(X,Y).

- See
  http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/recursion.html
**Negation as failure**

- Prolog lacks a direct way of proving that a proposition is false.
  
  \[
  \text{ancestor(george6, rahmEmanuel).}
  \]

- Because it lacks world knowledge. This is called the "closed world assumption".

- Instead, it just fails to show that it's provable.

**Warning:** It may report failure (acceptable), but it may get stuck in an infinite process (bad!).

---

**Lists**

- So far we've used only simple arguments (constants and variables) to our predicates.

- Prolog also handles *lists* enclosed in square brackets with comma separators

  \[
  \text{[superior, michigan, huron, erie, ontario]}
  \]

---

**Decomposing Lists for processing**

- Similar to Lisp and Clojure
  
  - The first element is called the **Head** (like Lisp `car` or `first`)

  \[
  \text{superior}
  \]

  - A list of remaining elements is the **Tail** (like Lisp `cdr` or `rest`)

  \[
  \text{[michigan, huron, erie, ontario]}
  \]

  - Processing a list often consists of
    - Doing something to/with/on the Head
    - Recursive invocation on the Tail

---

**Prolog versus Lisp**

(i.e. backward versus forward-chaining search emphasis)

- For reasons that aren't altogether clear:
  
  - A.I. researchers in America prefer Lisp and Lisp-like languages (Scheme, Clojure)
  
  - A.I. researchers in Europe and Asia prefer Prolog and Prolog-like languages.

- But they will use either approach when the problem clearly calls for it.

- Serious A.I. practitioners should feel comfortable with both.