Outline for today

- Quick review
- Equality and unification
- How Prolog searches for answers

Quick review

- Atoms bart 'Mr. Burns'
- Variables X Y Z
- Predicates p(t₁,...,tₙ)
- Terms
- Facts father(homer, bart).
- Goals p(t₁,...,tₙ), ..., q(t₁',...,tₙ').
- Rules p(ts) :- q(ts'), ..., r(ts'').

Infix operators

- Prolog has built-in constants and infix operators
- Examples:
  - Equality: t = u (or = (t,u))
  - Pairing: (t,u) (or , (t,u))
  - Empty list: []
  - List concatenation: [X|Y] (or .(X,Y))
- You can also define your own infix operators!
General observations

- Prolog is **untyped**
  - everything is a "term"
- Prolog is **declarative**
  - "predicates" with side effects, such as print, are the exception, not the rule
- Prolog does **not** have explicit control flow constructs (while, do)
  - the search strategy allows us to simulate iteration
  - but this is not usually the best way to program
- Therefore, try to **forget what you already know** from other languages

Terms

- Also have...
  - **Numbers**: 1 2 3 42 -0.12345
  - Additional **constants** and **infix operators**
- More on these later.

Unification (I)

- The equation \( t = u \) is a basic goal
  - with a special meaning
- What happens if we ask:
  - \(?- X = c\)
  - \(?- f(X, g(Y, Z)) = f(c, g(X, Y))\)
  - \(?- f(X, g(Y, f(X))) = f(c, g(X, Y))\)
- And how does it do that?
?- X = c.
X=c
yes
?- f(X,g(Y,Z)) = f(c,g(X,Y)).
X=c
Y=c
Z=c
yes
Unification (II)

?- X = c.
X=c
yes

?- \( f(X, g(Y,Z)) = f(c, g(X,Y)) \).
X=c
Y=c
Z=c
yes

?- \( f(X, g(Y,f(X))) = f(c, g(X,Y)) \).
no

Unification (III)

• A substitution is a mapping from variables to terms
  • \( X_1=t_1, \ldots, X_n=t_n \)
• Given two terms \( t \) and \( u \)
  • with free variables \( X_1 \ldots X_n \)
• a unifier is a substitution that makes \( t \) and \( u \) equal

Example (I)

\( f(X, g(Y,Z)) = f(c, g(X,Y)) \)
Example (I)

\[ f(X, g(Y, Z)) = f(c, g(X, Y)) \]
Example (I)

\[ f(X, g(Y, Z)) = f(c, g(X, Y)) \]

Example (II)

\[ f(X, g(Y, f(X))) = f(c, g(X, Y)) \]
Example (II)

\[ f(X, g(Y, f(X))) = f(c, g(X, Y)) \]
Example (II)

\[ f(X, g(Y, f(X))) = f(c, g(X, Y)) \]

\[ X = c \]
\[ Y = c \]
\[ Y = f(X) \]
\[ f(X) = c?? \]
Robinson's Algorithm (I)

- Consider a general unification problem
  \[ t_1 = u_1, \ t_2 = u_2, \ldots, \ t_n = u_n \]
- Reduce the problem by decomposing each equation into one or more "smaller" equations
- Succeed if we reduce to a "solved form", otherwise fail

Robinson's Algorithm (II)

- Two constants unify if they are equal.
  \[ c = c, \ P \rightarrow P \]
  \[ c = d, \ P \rightarrow \text{fail}. \]

Robinson's Algorithm (III)

- Two function applications unify if the head symbols are equal, and the corresponding arguments unify.
  \[ f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n), \ P \rightarrow \]
  \[ t_1 = u_1, \ldots, t_n = u_n, \ P \]
- Must have equal numbers of arguments
  \[ f(\ldots) = g(\ldots), \ P \rightarrow \text{fail} \]
  \[ f(\ldots) = c, \ P \rightarrow \text{fail}. \]

Robinson's Algorithm (IV)

- Otherwise, a variable \( X \) unifies with a term \( t \) provided \( X \) does not occur in \( t \).
  \[ X = t, \ P \rightarrow P[t/X] \]
  (occurs-check: \( X \) must not be in \( \text{Vars}(t) \))
- Proceed by substituting \( t \) for \( X \) in \( P \).
Occurs check

- What happens if we try to unify X with something that contains X?
  
  \[ ?- X = f(X). \]

- Logically, this should fail
  
  - there is no (finite) unifier!

- Most Prolog implementations skip this check for efficiency reasons
  
  - can use `unify_with_occurs_check/2`

Execution model

- The query is run by trying to find a solution to the goal using the clauses
  
  - Unification is used to match goals and clauses
  
  - There may be zero, one, or many solutions
  
  - Execution may backtrack

- Formal model called SLD resolution
  
  - which you'll see in the theory lectures

Depth-first search (I)

- Idea: To solve atomic goal A,
  
  - If B is a fact in the program, and \( \theta(A) = \theta(B) \), then return answer \( \theta \)

  Else, if \( B \leftarrow G_1, \ldots, G_n \) is a clause in the program, and \( \theta \) unifies \( A \) with \( B \), then solve \( \theta(G_1) \ldots \theta(G_n) \)

  Else, give up on this goal.

  - Backtrack to last choice point

- Clauses are tried in declaration order

- Compound goals are tried in left-right order

Depth-first search (II)

- Prolog normally tries clauses in order of appearance in program.

- Assume: `foo(a). foo(b). foo(c).`

- Then:
  
  \[ ?- foo(X). \]

  \[ \text{foo(X)} \]
Depth-first search (II)

- Prolog normally searches for clauses in order of appearance in database.
- Assume: `foo(a). foo(b). foo(c).
- Then:
  ```prolog
  X = a
  X = b;
  X = c;
  no
  ```
  done
  done
  done

James Cheney
Logic Programming
September 22, 2014
Depth-first search (III)

• Prolog backtracks to the last choice point if a subgoal fails.
• Assume: \( \text{bar}(b). \text{bar}(c). \text{baz}(c) \).
• Then:
  \[ ?- \text{bar}(X), \text{baz}(X). \]

 X=b
 baz(b)

 Depth-first search (III)

• Prolog backtracks to the last choice point if a subgoal fails.
• Assume: \( \text{bar}(b). \text{bar}(c). \text{baz}(c) \).
• Then:
  \[ ?- \text{bar}(X), \text{baz}(X). \]

 X=b
 baz(b)
Depth-first search (III)

- Prolog backtracks to the last choice point if a subgoal fails.
- Assume: \( \text{bar}(b) \), \( \text{bar}(c) \), \( \text{baz}(c) \).
- Then:
  \[ ?- \text{bar}(X), \text{baz}(X). \]

\[ \text{bar}(X), \text{baz}(X) \]
\[ X=b \]
\[ \text{baz}(b) \]
\[ \text{X=b} \]
\[ \text{baz}(b) \]

- Prolog backtracks to the last choice point if a subgoal fails.
- Assume: \( \text{bar}(b) \), \( \text{bar}(c) \), \( \text{baz}(c) \).
- Then:
  \[ ?- \text{bar}(X), \text{baz}(X). \]

\[ \text{bar}(X), \text{baz}(X) \]
\[ X=b \]
\[ \text{X=b} \]
\[ \text{baz}(b) \]

\[ \text{X=c} \]
\[ \text{baz}(c) \]

\[ \text{baz}(b) \]

\[ \text{X=b} \]
\[ \text{baz}(b) \]
\[ \text{X=c} \]

\[ \text{done} \]
Depth-first search (III)

- Prolog **backtracks** to the last choice point if a subgoal fails.
- Assume: bar(b). bar(c). baz(c).
- Then:

  ?- bar(X),baz(X).
  
  X = c;
  no

"Generate and test"

- Common Prolog programming idiom:

  find(X) :- generate(X), test(X).

  where test(X) **checks if X is a solution**

  generate(X) **searches for solutions**

  - Can use to constrain (infinite) search space
  - Can use different generators to get different search strategies besides depth-first

Limitations of depth-first search

- Recursion needs to be handled carefully to avoid loops
- Rule order and goal order matter
- More in next lecture
- Not complete "in practice"

  - legitimate answers may be missed due to loops

Other search strategies

- Breadth-first search / iterative deepening
  - Explore all alternatives, interleaved
  - Price: memory overhead
- Bottom-up (forward chaining)
  - Compute all possible answers derivable from facts & rules
  - Only viable for "Datalog" programs with "flat" data (only constants and variables)
  - Supported by commercial tools for big data (LogicBlox, DLV)
Next time

• Recursion
• Lists, trees, data structures
• Further reading: LPN ch. 2
• Tutorial #1 will be up soon