Prolog: facts

- A fact is a clause with an empty body

Syntax
- `<head>`.

What makes a fact a fact?

Examples
- Exams: `exams.`
- Assignments: `assignments.`
- Taxes: `taxes.`
- The earth is round: `round(earth).`
- The sky is blue: `blue(sky).`
- The sun is hot: `hot(sun).`
- Mary is a female: `female(mary).`
- Beethoven lived between 1770 & 1827: `person(beethoven,1770,1827).`

Prolog: rules

- A rule in Prolog is in a full horn clause format:

  \[ c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge ... \wedge h_n \]

Syntax:

- If I know that all those relations (those in the body) hold, then I also know that this LHS relation (in the head) holds.

Examples:
- If there is smoke there is fire
  \[ fire \leftarrow smoke. \]
- If the course is boring, I leave
  \[ leave(i) \leftarrow boring(course). \]
- Joe is going to kill the teacher if he fails CSC324.
  \[ kills(joe, X) \leftarrow fails(joe,csc324), teaches(X,csc324). \]
Prolog: rules – cont’d

- Examples:
  - X is female if X is the mother of anyone.
  
    female(X) :- mother(X,_). % avoid singleton variables by using _.
  
  - X is the sister of Y, if X is female and X’s parents are M and F, and Y’s parents are M and F
    
    sister_of(X,Y):- female(X),parents(X,M,F),parents(Y,M,F).

  % in general, how we interpret the rule in first-order logic (predicate logic)?

- When to use rules?
  - Use rules to say that a particular fact depends on a group of facts.
  - Use rules to deduce new facts from existing ones.

- Rules of rules:
  - The head of the rule consist of at most one predicate
  - The body of the rule is a finite sequence of literals separated by ‘,’ (which means conjunction and )
  - Rules always end with a period “.”

Prolog: queries – cont’d

- Examples

  - composer(beethoven,1770,1827).
    - is it true that beethoven was a composer
    
    who lived between 1770 and 1827

  - owns(john,book).
    - is it true that john owns a book?
    
    (simpler: does john own a book?)

  - owns(john,X).
    - is it true that john owns something?
    
    (simpler: does john own something?)

Prolog: queries

- A query is a clause with an empty head.

  \[ \text{\texttt{\textless{}h_1 \& h_2 \& h_3 \& \ldots \& h_n}} \]

- Syntax

  - `<body>`.
    
    Try to prove that `<body>` is true

    The goal is represented to the interpreter as a question.

- Examples

  - round(earth).
    - Is it true that the earth is round?
    
    % (or simpler than that: is the earth round?)

  - round(X).
    - Is it true that there are entities which are round?
    
    % (or simpler than that: what entities are round?)

Prolog: simple types - constants

- There are two types of constants: atoms and numbers.

- Atoms:
  - Alphanumeric atoms: alphabetic sequence starting with a lower case letter
    
    - E.g.: apple a1 apple_cart

  - Special atoms
    
    - E.g. ! ; [ ]

  - Symbolic atoms: sequence of symbolic characters
    
    - E.g. & < > * - +

  - Quoted atoms: sequence of characters surrounded by single quotes
    
    - Can make anything an atom by enclosing it in single quotes.
    
    - E.g ‘apple’ ‘hello world’

- Numbers:

  - Integers and Floating Point numbers
    
    - E.g. 0 1 9821 -10 1.3 -1.3E102
Prolog: complex types - structures

- Recall: what’s a functional term?
  
  \[ \text{functor}(\text{some-parameters}) \quad \text{e.g.\ office(mary)} \]

- We can construct complex data structures using nested functional terms.
  
  - Represents a statement about the world

- Example:
  
  - A person has; name: first name, last name - birth date: day, month, year &
  occupation

\[ \text{person(name(michael, jordan), birth\_date(17, february, 1963), occupation('NBA player'))} \]

Prolog: complex types - structures

**Database:**

- owns(john, car(red, corvette))
- owns(john, cat(black, siamese, sylvester))
- owns(elvis, copyright(song,"jailhouse rock"))
- owns(tolstoy, copyright(book,"war and peace"))
- owns(elvis, car(red, cadillac))

**Query:**

"Retrieve everything that John owns."

i.e., Find \(x\) such that \(\text{owns(john,} x\text{)}\) is true.

\[ \text{answers: } x = \text{car(red, corvette)} \]
\[ x = \text{cat(black.siamese.sylvester)} \]

**Query:**

"Retrieve the colour and make of John's car."

i.e., \(\text{owns(john,}\ car(\text{ Colour, Make}))\)

\[ \text{answer: Colour = red} \]
\[ \text{Make = corvette} \]

Prolog: complex types - structures

**Database:**

- owns(john, car(red, corvette))
- owns(john, cat(black, siamese, sylvester))
- owns(elvis, copyright(song,"jailhouse rock"))
- owns(tolstoy, copyright(book,"war and peace"))
- owns(elvis, car(red, cadillac))

**Query:**

"Who owns a red car?"

i.e., Find values for who so that
\[ \exists x, y \text{ owns(who, car(red, make))} \] is true.

\[ \text{answers: Who = john} \]
\[ \text{Who = elvis} \]
**Prolog: an example**

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>likes(eve, pie). food(pie).</td>
</tr>
<tr>
<td>likes(al, eve). food(apple).</td>
</tr>
<tr>
<td>likes(eve, tom). person(tom).</td>
</tr>
<tr>
<td>likes(eve, eve).</td>
</tr>
</tbody>
</table>

Query:

?-likes(al, pie).
- no

?-likes(al, eve).
- yes

?-likes(eve, al).
- no

?-likes(person, food).
- no

**Prolog: example – cont’d**

<table>
<thead>
<tr>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>likes(eve, pie). food(pie).</td>
</tr>
<tr>
<td>likes(al, eve). food(apple).</td>
</tr>
<tr>
<td>likes(eve, tom). person(tom).</td>
</tr>
<tr>
<td>likes(eve, eve).</td>
</tr>
</tbody>
</table>

?-likes(A,B).
A=eve,B=pie ; A=al,B=eve ; ...  
?-likes(D,D).
D=eve ; no

?-likes(eve,W), person(W).  
W=tom

?-likes(al,V), likes(eve,V).  
V=eve ; no

**Prolog: proof procedure**

- Two main processes:
  - Unification
  - Top-down reasoning

**Prolog: unification**

- First step in proof procedure

- Prolog tries to satisfy a query by **unifying** it with some conclusion and see if it is true!

- Process of finding these suitable "assignments" of values to variables is called **unification**
  - It is really a process of pattern matching to make statements identical
  - Somewhat similar to variable bindings in imperative world and to pattern matching in Scheme.
**Prolog: unification – cont’d**

- **Rules of unification:**

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>example</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>free var.</td>
<td>X</td>
<td>X=4</td>
</tr>
<tr>
<td>bound variable</td>
<td>free variable</td>
<td>X, Y</td>
<td>Y gets the value of X</td>
</tr>
<tr>
<td>free variable</td>
<td>bound variable</td>
<td>X, Y</td>
<td>X gets the value of Y</td>
</tr>
<tr>
<td>bound variable</td>
<td>constant</td>
<td>X</td>
<td>“b” fails if X has a value different than “b”</td>
</tr>
<tr>
<td>compound object</td>
<td>compound object</td>
<td>f(X,Y)</td>
<td>X=2, Y=3</td>
</tr>
<tr>
<td>compound object</td>
<td>compound object</td>
<td>f(q(2,X),3)</td>
<td>f(P,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>succeeds if P is free, and P=q(2,X) . (and more possibilities)</td>
<td></td>
</tr>
</tbody>
</table>

- **Examples:**

| a(b, c, d, E) with x( ... ) | doesn’t unify: a and x differ |
| a(b, c, d, E)                | no: different # of args |
| a(b, c, d, E)                | no: b ≠ j |
| a(j, f, G, H)                | yes: by either { C ← f, G ← d, H ← E } |
| a(b, f, G, H)                | or { C ← f, G ← d, E ← H } |
| a(pred(X, j))                | yes: { X ← k } |
| a(pred(k, j))                | yes: { X ← k } |
| a(pred(X, j))                | yes: { B ← pred(X, j) } |

**Prolog: unification – cont’d**

- **Rules of unification:**

  - A constant unifies only with itself, it cannot unify with any other constant.
  - Two structures unify iff they have the same name, number of arguments and all the arguments unify.
  - Unification requires all instances of the same variable in a rule to get the same value.

- **Examples:**

  - Does p(X,X) unify with p(b,b) ?
  - Does p(X,X) unify with p(b,c) ?
  - Does p(X,b) unify with p(Y,Y) ?
  - Does p(X,Z,Z) unify with p(Y,Y,b) ?
  - Does p(b,b,X) unify with p(Y,Y,c) ?
    - To make the third arguments equal, we must replace X by c
    - To make the second argument equal, we must replace Y by b.
    - So, p(X,b,X) becomes p(c,b,c), and p(Y,Y,c) becomes p(b,b,c).
    - However, p(c,b,c) and p(b,b,c) are not syntactically identical.
Prolog: example 2

- Facts & rules:

  ![Graph Diagram]

  - link(a, b), link(b, c), link(a, d), link(d, c).
  - path(N, N).
  - path(L, M) :- link(L, X), path(X, M).

- Posing queries:

  Based on our logical encoding of the graph, we can then write queries:

  ?- path(a, c)
  yes
  ?- path(c, a)
  no
  ?- path(a, X), path(X, c)
  X = a
  X = b
  X = c
  X = d

  Notice that we didn’t write a graph traversal algorithm, and we didn’t hard code the set of questions we can ask in advance. We just define what a graph is...

Prolog: reasoning

- Given a set of facts and rules, we need a mechanism to deduce new facts and/or prove that a given rule is true or false or has no answer

- There are two techniques to do this:
  - Bottom-up reasoning
  - Top-down reasoning

Prolog: proof procedure - revisited

- Two main processes:
  - Unification
  - Top-down reasoning

    ![Diagram of proof procedure]

Bottom-up Reasoning

- Bottom-up (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  *e.g.*, Suppose the fact \( B \) and the rule \( A \leftarrow B \) are given. Then infer that \( A \) is true.

  **Example**

  Rule base:

  \[
  p(X, Y, Z) \leftarrow q(X), q(Y), q(Z).
  q(a1).
  q(a2).
  \ldots
  q(aN).
  \]

  Bottom-up inference derives \( n^3 \) facts of the form \( p(a_1, a_2, a_3) \):

  \[
  p(a1, a1, a1)
  p(a1, a1, a2)
  p(a1, a2, a3)
  \ldots
  \]

  So, \( A \) is proved
Prolog: top-down reasoning

- **Top-down** (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

  *E.g.*, Suppose the query is \( A \), and the rule \( A \leftarrow B \) is given. Then to prove \( A \), try to prove \( B \).

![Rule Base and Top-down Proof Diagram]

So, \( A \) is proved.

Prolog: top-down reasoning – cont’d

- **Multiple rules and multiple premises:**
  - A fact may be inferred by many rules
    *E.g.* \( E \leftarrow B \)
    \( E \leftarrow C \)
    \( E \leftarrow D \)
  - A rule may have many premises
    *E.g.* \( E \leftarrow B \land C \land D \)

- In top-down inference, such rules give rise to
  - Inference trees
  - Backtracking

Prolog: top-down reasoning – cont’d

- **Example:** *multiple premises*

  **Rule base:**
  
  1. \( A \leftarrow B_1 \land B_2 \)
  2. \( B_1 \leftarrow C_1 \land C_2 \)
  3. \( B_2 \leftarrow C_3 \land C_4 \)

  **Goal:** \( A \)

  **Rule (1):**

  \( A \leftarrow B_1 \land B_2 \)

  **Rule (2):**

  \( B_1 \leftarrow C_1 \land C_2 \)

  **Rule (3):**

  \( B_2 \leftarrow C_3 \land C_4 \)

  **Query:** Is \( A \) true?

  **Goal C1:** \( C_1 \land C_2 \)

  **Goal C2:** \( C_3 \land C_4 \)

  **Goal C3:** success

  **Goal C4:** success

  **Goal B1:** success

  **Goal B2:** success

  **Success:**

  So, goal \( A \) is proved. (all paths must succeed)

Prolog: top-down reasoning – cont’d

- **Example:** *multiple rules*

  **Rule base:**

  \( A \leftarrow B_1 \)

  \( B_1 \leftarrow C_1 \)

  \( B_1 \leftarrow C_2 \)

  \( B_2 \leftarrow C_3 \)

  \( B_2 \leftarrow C_4 \)

  **Goal:** \( A \)

  **Rule (1):**

  \( A \leftarrow B_1 \)

  **Rule (2):**

  \( B_1 \leftarrow C_1 \)

  **Rule (3):**

  \( C_3 \leftarrow C_4 \)

  **Query:** Is \( A \) true?

  **Goal C1:** fail

  **Goal C2:** fail

  **Goal C3:** fail

  **Goal C4:** success

  **Success:**

  So, goal \( A \) is proved. (only one path must succeed)
Prolog: backtracking

- Prolog uses this algorithm for proving a goal by recursively breaking goal down into sub-goals and try to prove these sub-goals until facts are reached.

- To satisfy a goal:
  - Try to unify with conclusion of first rule in database
  - If successful, apply substitution to first premise, try to satisfy resulting sub-goals
  - Then apply both substitutions to next sub-goal (premise), and so on...
  - If not successful, go on to the next rule in database
  - If all rules fail, try again (backtrack) to a previous sub-goal

Prolog: backtracking example 1

Rule base:

\[
p(X) : - q(X), r(X).
q(d). q(e). q(f). q(g).
r(e). r(g).
\]

Query: Find \( x \) such that \( p(x) \) is true.

\[
p(X)
q(X), r(X)
X=d \rightarrow r(d) \text{ fail}
X=e \rightarrow r(e) \text{ success (print "X=e")}
X=f \rightarrow r(f) \text{ fail}
X=g \rightarrow r(g) \text{ success (print "X=g")}
\]

Prolog: backtracking example 2

Rule base:

\[
p(X) : - q(X), r(X, Y), s(Y).
q(a). r(a, b). r(c, b). s(c).
q(c). r(a, c). r(c, c).
r(a, d).
\]

Query: Find \( x \) such that \( p(x) \) is true.

Prolog: backtracking example 3

Query: `- located_in(toronto, north_america)`

matches 1 under x=toronto

matches 0 under x=toronto

matches 18 under x=toronto, usa

matches 5 under x=toronto, georgia

No Matches

Fail
Top-down vs. Bottom-up Reasoning

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (e.g. Coral)

- Each has its own advantages and disadvantages:
  - Bottom-up may generate many irrelevant facts
  - Top-down may explore many lines of reasoning that fail.

- Top-down and bottom-up inference are logically equivalent
  - i.e. they both prove the same set of facts.

- However, only top-down inference simulates program execution
  - i.e. execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc...