Today

- Logic Programming
- Prolog I

LP: resolution in predicate calculus

- We would like to infer new propositions (e.g., facts) from some existing set of propositions.
- An inference rule that can be applied atomically is called a resolution.
  - E.g. Given: \( P_1 \land P_2 \rightarrow Q_1 \land Q_2 \)
    - Alternatively: \( T \land P_2 \rightarrow Q_1 \land T \)
    - New fact: \( Q_1 \land P_2 \)
- Resolution gets more complex if variables/values are involved:
  - To use resolution with variables, we will need to find values for variables that allow matching to proceed.
  - E.g. Given: \( F(X,Y) \land P_2(Y,X) \)
    - \( Q_1(\text{foo}) \land F(\text{foo}, \text{bar}) \)
    - New fact: \( Q_1(\text{foo}) \land P_2(\text{bar}, \text{foo}) \)

LP: horn clause

- Logic programming is heavily based on horn clauses:
  - \( c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \)
  - Antecedents \( (h_1, h_2, h_3, \ldots, h_n) \) are conjunction of zero or more conditions which are atomic constructs in predicate logic.
  - Consequent \( c \) is an atomic construct in predicate logic.
- Meaning of a horn-clause:
  - The consequent is true if the antecedents are all true.
    - \( c \) is true if \( h_1, h_2, h_3, \ldots \) are all true.
- A horn clause can capture most, but not all, logical statements/implications, why?
Many non-horn rules can be transformed to horn form using one of two methods:

- Logical equivalence
- Skolemization

Logical equivalence:
- Uses the following logical laws:
  - Negation: \( \neg \neg A \equiv A \)
  - De Morgan’s Law:
    - \( \neg (A \lor B) \equiv \neg A \land \neg B \)
    - \( \neg (A \land B) \equiv \neg A \lor \neg B \)
  - Distributive Property:
    - \( A \land (B \lor C) \equiv (A \land B) \lor (A \land C) \)
    - \( A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \)
  - Absorption Law:
    - \( A \lor (A \land B) \equiv A \)
    - \( A \land (A \lor B) \equiv A \)
  - Implication Laws:
    - \( A \rightarrow B \equiv (A \land B) \lor (A \land \neg B) \)

- Examples:
  - \( \neg A \lor \neg B \equiv A \land \neg (A \lor B) \)
  - \( A \land (B \lor C) \lor \neg B \equiv A \land (B \lor C) \land \neg B \)

Horn clause

- What are we going to do about quantifiers?

Skolemization:
- Variables bound by existential (\( \exists X \)) quantifiers which are not inside the scope of universal quantifiers can simply be replaced by constants:
  - \( \exists X \) becomes c

- When the existential quantifier (\( \exists X \)) is inside a universal quantifier (\( \forall Y \))
  - \( \forall X \exists Y \) becomes \( \forall X \) [\( \exists Y \) variable]

If we also have a function \( f \) which is defined on a universe of discourse, we can simply substitute \( f(X) \) for \( X \) in the expression:

\[ \forall X \] becomes \( \forall X \) [\( f(X) \)]
LP: specifying non-horn rules – cont’d

- Skolemization:
  - Non-horn formulas like (∃X) A(X) can be converted to horn-clause by introducing a skolem constant and/or skolem function. The resulting clause is almost the same thing.

- Why does skolemization work?
  - We only need ∃X because we don’t have a name for X. By creating artificial names (skolem names), we can eliminate many ∃’s and convert many formulas to horn clause.

LP: horn clause made easy!

- Horn clause can include more complex terms:
  - p(X) ← q(X,Y) A(1,Y) A s(X,Y)
  - p(X) ← ……
  - p(X) ← ……
  - p(X) ← ……

- We can assume the following when writing horn-clauses:
  - p is the program name
  - q, s, a are the subprogram names
  - X is a parameter of the program
  - Y is a local variable

Prolog:

- The first and most popular logic programming language
  - Invented by Alain Colmerauer and Philippe Roussel at the University of Aix-Marseille in 1971 (France)

- Characteristics:
  - Is very weakly typed
  - Has no data abstraction
  - Has no functional abstraction!
  - Has no mutable state
  - Has no explicit control flow

- So, how do you program?
  - Load facts/rules into interpreter
  - Make queries to see if a fact is:
    - in the knowledge-base or
    - can be implied from existing facts or rules

- Prolog is really an engine to prove theorems
Prolog: data types – quick intro

- **Simple**
  - **Constants**:
    - Numbers: integer, floating point,...
    - Atoms: alphabetic sequence starting with a lower case letter (e.g. apple)
  - **Variables**:
    - Variables start with capital letters or underscore

- **Complex**
  - Lists
  - Structures

Prolog: horn clauses

- **Recall**
  \[ c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \]

- **Syntax**:
  - You can conclude that `<head>` is true, if you can prove that `<body>` is true
  - The symbol `\leftarrow` is read as `if`

- **3 types of clauses**:
  - Facts
  - Rules
  - Queries

Prolog: facts

- A fact is a clause with an empty body

- **Syntax**
  \[ \text{<head>} \]

- **What makes a fact a fact?**

- **Examples**
  - Exams
  - Assignments
  - Taxes
  - The earth is round.
  - The sky is blue.
  - The sun is hot.
  - Mary is a female.
  - Beethoven lived between 1770 & 1827.

Prolog: facts – cont’d

- **Facts about facts**:
  - Full stop `.` at the end of every fact.
  - The number of arguments in a fact is called arity.
  - E.g. `female(mary)` is an instance of `female/1` (functor `female`, arity 1)
    - Facts with different number of arguments are distinct
      - E.g. `female(mary)` is different from `female(mary)`. 
Prolog: rules

• A rule in Prolog is a full horn clause:
  \[ c \leftarrow h_1 \land h_2 \land h_3 \ldots \land h_n \]

• Syntax:
  \[ \text{rel}_1 \leftarrow \text{rel}_2 \land \text{rel}_3 \land \ldots \land \text{rel}_n \]

  – 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
    \[ \text{fire} \leftarrow \text{smoke} \]
  – If the course is boring, I leave
    \[ \text{leave}(i) \leftarrow \text{boring}(\text{course}) \]
  – Joe is going to kill the teacher if he fails CSC324.
    \[ \text{kills}(\text{joe}, X) \leftarrow \text{fails}(	ext{joe}, \text{csc324}), \text{teaches}(X, \text{csc324}) \]

• 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 \(\land\) or
    conjunction (and)
  – Rules always end with a period `.`

Prolog: queries

• A query is a clause with an empty head.
  \[ \text{body} \]

• Syntax
  \[ \text{?} \text{body} \]

  – Try to prove that \text{body} is true
  – The goal is represented to the interpreter as a question.

• Examples
  \[ \text{?round(earth).} \] - is it true that the earth is round?
  (or simpler than that: is the earth round?)

  \[ \text{?round}(X). \] - is it true that there are entities which are round?
  (or simpler than that: what entities are round?)

Prolog: rules – cont’d

• Examples:
  – X is female if X is the mother of anyone.
    \[ \text{female}(X) \leftarrow \text{mother}(X, \_). \]
  – 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
    \[ \text{sister}_o(X, Y) \leftarrow \text{female}(X), \text{parents}(X, M, F), \text{parents}(Y, M, F). \]

• 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 \(\land\) or
    conjunction (and)
  – Rules always end with a period `.`

Prolog: queries – cont’d

• Examples
  \[ \text{?composer(beethoven, 1770, 1827).} \] - is it true that beethoven was a
  composer who lived between 1770 and 1827

  \[ \text{?owns(john, book).} \] - is it true that john owns a book?
  (simpler: does john own a book?)

  \[ \text{?owns(john, X).} \] - is it true that john owns something?
  (simpler: does john own something?)
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. !, apple, car
  - Special atoms
    - 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: simple types - variables

- Variables start with **capital letters** or **underscore**

- There are **no global variables** (assert and retract, will see them later...)

- **Instantiated vs. un-instantiated**:
  - if the object a variable stands for is already determined, var is **instantiated**
  - if the object a variable stands for is not yet determined, var is **un-instantiated**

- An instantiated variable in Prolog cannot change its value

- Variables are limited in scope to the clause they appear in (local vars)
  - E.g. grandParent(X,Y) :- parent(X,Z), parent(Y,Z). % The Xs here are the same var
  - Parent(X,Y) :- mother(X,Y). % But not the same as those here

- There is a special anonymous variable “_” which is used to denote “don’t care”
  - E.g. Parent(X) :- mother(X,_)
  - Note that every use of _ is considered a separate variable

Prolog: example 1

Facts:

```
likes(eve, pie).    food(pie).
likes(al, eve).     food(apple).
likes(eve, tom).    person(tom).
likes(eve, eve).
```

```
?-likes(al, pie).
   no
?-likes(al, eve).
   yes
?-likes(eve, al).
   no
?-likes(person, food).
   no
```

?-likes(al, Who).
   My mom?
?-likes(eve, W).
```

```
?-likes(al, eve).
   pie
?-likes(al, eve).
   tom
?-likes(al, eve).
   eve
```

Prolog: example 1 – cont’d

Facts:

```
likes(eve, pie).    food(pie).
likes(al, eve).     food(apple).
likes(eve, tom).    person(tom).
likes(eve, eve).
```

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

```
?-likes(eve, W).    person(W).
   Tom
?-likes(al, V), likes(eve, V).
   V = eve ; no
```
Prolog: proof procedure

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

Prolog: unification – cont’d

- Rules of unification:

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>Sample</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>free var.</td>
<td>X</td>
<td>X=Y</td>
</tr>
<tr>
<td>bound var.</td>
<td>free var.</td>
<td>X</td>
<td>Y has the value of X</td>
</tr>
<tr>
<td>free var.</td>
<td>bound var.</td>
<td>X</td>
<td>Y gets the value of Y</td>
</tr>
<tr>
<td>bound var.</td>
<td>constant</td>
<td>X</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>compound</td>
<td>compound</td>
<td>f(X,Y)</td>
<td>X=Y, Y=3</td>
</tr>
<tr>
<td>compound</td>
<td>compound</td>
<td>f(q(X,Y),p)</td>
<td>succeeds if p is free, and P=q(X) and p=q(X)</td>
</tr>
<tr>
<td>compound</td>
<td>compound</td>
<td>f(X)</td>
<td>q(X)</td>
</tr>
</tbody>
</table>

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 ML.

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.
Prolog: unification – cont’d

- Examples:
  - `a(b, c, d, e)` doesn’t unify: `a` and `x` differ
  - `a(b, c, d, e)` no: different # of args
  - `a(j, f, g, h)` no: `j` ≠ `j`
  - `a(b, c, d, e)` yes: by either `[C → f, G → d, H → E]`
    or `[C → f, G → d, H → H]`
  - `a(pred(X, j))` yes: `[X → k]`
  - `a(pred(X, j))` yes: `[X → j]`
  - `a(b)` yes: `[b → pred(X, j)]`

Prolog: example 2

- Facts & rules:
  - `link(a, b), link(b, c), link(a, d), link(d, c), path(N, N), path(J, M), path(J, X), path(X, M)`

- Posing queries:
  - Based on our logical encoding of the graph, we can then write queries.
    - `?- path(a, c)`
      yes
    - `?- path(a, 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: proof procedure - revisited

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

- Prolog: proof procedure -- revisited
  - [Diagram showing proof procedure with rules and facts]
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: 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 $D$ and the rule $A \leftarrow B$ are given. Then infer that $A$ is true.

Example

<table>
<thead>
<tr>
<th>Rule base:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(a, b, c)$</td>
</tr>
<tr>
<td>$q(x)$</td>
</tr>
<tr>
<td>$\vdots$</td>
</tr>
<tr>
<td>$\vdots$</td>
</tr>
<tr>
<td>$p(a, b, c)$</td>
</tr>
</tbody>
</table>

Bottom-up inference derives $A$ facts of the form $p(a, b, c)$:

- $p(a, b, c)$
- $p(a, b, c)$
- $p(a, b, c)$

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 $B$.

Example

<table>
<thead>
<tr>
<th>A rule base:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \leftarrow B$</td>
</tr>
<tr>
<td>$B \leftarrow C$</td>
</tr>
<tr>
<td>$C$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A top-down proof:</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal $A$</td>
</tr>
<tr>
<td>rule (1)</td>
</tr>
<tr>
<td>goal $B$</td>
</tr>
<tr>
<td>rule (2)</td>
</tr>
<tr>
<td>goal $C$</td>
</tr>
<tr>
<td>rule (3)</td>
</tr>
<tr>
<td>success</td>
</tr>
</tbody>
</table>

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:
    - Rule 1: $A \leftarrow B_1 \land B_2$
    - Rule 2: $B_1 \leftarrow C_1 \land C_2$
    - Rule 3: $B_2 \leftarrow C_3 \land C_4$
    - Goal A

  - Query: Is A true?

  - Goal $B_1$:
    - Rule 1: $C_1 \land C_2$
    - Rule 2: $C_3 \land C_4$
  - Goal $B_2$:
    - Rule 3: $C_3 \land C_4$

  Success

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

**Prolog: backtracking**

- **Example: multiple rules**

  - Rule base:
    - Rule 1: $A \leftarrow B_1 \land B_2$
    - Rule 2: $B_1 \leftarrow C_1 \land C_2$
    - Rule 3: $B_2 \leftarrow C_3 \land C_4$
    - Goal A

  - Query: Is A true?

  - Goal $B_1$:
    - Rule 1: $C_1 \land C_2$
    - Rule 2: $C_3 \land C_4$
  - Goal $B_2$:
    - Rule 3: $C_3 \land C_4$

  Success

  So, goal A is proved. (only one path must succeed)

**Prolog: backtracking example 1**

- **Rule base:**
  - $p(X) :\leftarrow q(X), r(X)$,
  - $q(X), q(Y), q(Z), q(G)$,
  - $r(X), r(Y)$.

  - Query: Find X such that $p(X)$ is true.

  - p(X)
    - q(X), r(X)
      - f fail
      - g success (print "X=g")
  - r(Y) fail
  - r(Z) fail
  - r(G) fail

  So, goal A is proved. (only one path must succeed)
Prolog: backtracking example 2

Rule base:
\[ p(X) :- q(Q), z(X,Y), z(Y,Z). \]
\[ q(a), z(a,b), z(c,b), z(c,A). \]

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

Prolog: backtracking example 3 – cont’d

1. located_influenza, georgia.
2. located_influenza, campofranco.
3. located_influenza, california.
4. located_influenza, ontario.
5. located_influenza, usa.
6. located_influenza, canada.
7. located_influenza, north_america.
8. located_influenza, canada.
9. located_illness, north_america.
10. located_illness, canada.

\[ \Rightarrow \text{located_influenza, north_america}. \]
Prolog: 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.

Prolog: complex types - structures

Database:
- owns(john, car(red, corvette))
- owns(john, car(black, sierra, sylvester))
- owns(elvis, copyright(english, "jailhouse rock"))
- owns(olivier, copyright(french, "war and peace"))
- owns(elvis, car(red, cadillac))

Query:
- "Retrieve everything that John owns."
  i.e. Find X such that: owns(john, X) is true.
  answers: X = car(red, corvette)
  X = car(black, sierra, sylvester)

Query:
- "Retrieve the colour and make of John's car."
  i.e. owns(john, car(colour, make))
  answer: Colour = red
  Make = corvette

Prolog: complex types - structures

- Recall: what’s a function term? e.g. woman(marry)

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

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

Prolog: complex types - structures

Same Database:
- owns(john, car(red, corvette))
- owns(john, car(black, sierra, sylvester))
- owns(elvis, copyright(english, "jailhouse rock"))
- owns(olivier, copyright(french, "war and peace"))
- owns(elvis, car(red, cadillac))

Query:
- "Who owns a red car?"
  i.e. Find values for X so that
  owns(who, car(red, X)) is true.
  answers: Who = John
  Who = Elvis
Prolog: complex types - structures

Same Database:
  owns(john, car(red, corvette))
  owns(john, cat(black, flounder, sylvester))
  owns(erin, copyright(song, "jailhouse rock"))
  owns(erin, copyright(book, "war and peace"))
  owns(erin, car(red, cadillac))

Query: "Who owns a copyright?"
  i.e., Find values for who so that
    Ex,Y owns(who, copyright(X,Y)) is true.
  answers: who = erin
            who = erin