CASE STUDY
LOGIC PROGRAMMING
OUTLINE

- Preliminary Concepts
- Horn Clauses
- Logic Programming in Prolog
- Prolog Program Elements
- Practical Aspects of Prolog
- Prolog Examples
  - Solving Word Puzzles
  - Natural Language Processing
COMPUTATION VS. DEDUCTION

- **Compute:**
  - An expression + rules (the program) $\Rightarrow$ result

- **Deduce:**
  - A conjecture + rules (axioms/inference rules) $\Rightarrow$ a proof

- Logic Programming unites these two:
  - If we fix a strategy for proof search, then deduction can be considered a computation
  - Strategy $==$ algorithm in logic
**Proof System**

- Recall Curry-Howard Isomorphism:
  - Logic == Type system

- Inference rule:

  \[ \frac{J_1 \ldots J_k}{J} \]

  - premises
  - conclusion

  **E.g.,**

  \[
  \frac{\text{even}(z)}{\text{even}(z)} \quad \frac{\text{even}(N)}{\text{even}(s(s(N)))}
  \]

  - (even-z)
  - (even-s)

- Deduction:

  \[
  \frac{\text{even}(z)}{\text{even}(z)} \quad \frac{\text{even}(s(z))}{\text{even}(s(s(z)))}
  \]

  - (even-z)
  - (even-s)

  \[
  \frac{\text{even}(s(s(z)))}{\text{even}(s(s(s(s(z)))))}
  \]

  - (even-s)
PROOF SEARCH

- Search for the right rule to apply at each step.
- Two strategies:
  - Backward reasoning (goal-directed): from conjecture (a.k.a. goal) to axioms.
  - Forward reasoning: from axioms to conjecture.

- Negation as failure

\[
\begin{align*}
even(s(z)) \quad & \quad & \text{even}(s(z)) \text{ fails!} \\
even(s(s(s(z)))) \quad & \quad & \text{even}(s(s(s(z)))) \text{ is false!}
\end{align*}
\]
ANSWER SUBSTITUTION

- Previous example tells us if a number is even or not (yes if there is a proof, and no otherwise).
- LP also compute values
- Define natural number addition:

\[
\begin{align*}
&\text{plus}(z, N, N) \quad \text{(plus-z)} \\
&\text{plus}(s(N), M, s(P)) \quad \text{(plus-s)}
\end{align*}
\]

- Consider the conjecture (goal): \( \text{plus} \ (s(z), \ s(z), \ R) \).
  - Not only prove that this goal is true.
  - Also want to know the value of R.
  - Search not only constructs a proof, but also searches for a value R that makes the goal hold.
**Answer Substitution**

- Do the deduction on two rules:

  \[
  \begin{align*}
  &\frac{\text{plus}(z, s(z), s(z))}{\text{plus}(z, s(z), P)} \\
  &\frac{\text{plus}(z, s(z), P)}{\text{plus}(s(z), s(z), R)}
  \end{align*}
  \]
  
  (plus-z) and \( P = s(z) \)
  
  (plus-s) and \( R = s(P) \)

- Substitute \( s(z) \) for \( P \), we get \( R = s(s(z)) \), i.e.,

  \[
  \begin{align*}
  &\frac{\text{plus}(z, s(z), s(z))}{\text{plus}(z, s(z), s(z))} \\
  &\frac{\text{plus}(z, s(z), s(z))}{\text{plus}(s(z), s(z), s(s(z)))}
  \end{align*}
  \]
**Backtracking**

- When a goal matches the conclusion of more than one rule: we reach a *choice point*.
- At the choice point, we pick a rule and attempt the proof.
- If that attempt fails, we go back to the most recently choice point, pick another rule.
- This process is called *backtracking*.
Consider goal: $\text{plus}(M, s(z), s(s(z)))$

- Computing $M = 2 - 1$

\[
\begin{align*}
\text{plus}(M_1, s(z), s(z)) & \quad \text{plus-s) and } M = s(M_1) \\
\text{plus}(M, s(z), s(s(z))) & \quad \text{(plus-s) and } M = s(M_1)
\end{align*}
\]

Both rule (plus-s) and rule (plus-z) can fire.

If we pick (plus-s):

Fail!

\[
\begin{align*}
\text{plus}(M_2, s(z), z) & \quad \text{(plus-s) and } M_1 = s(M_2) \\
\text{plus}(M_1, s(z), s(z)) & \quad \text{(plus-s) and } M = s(M_1) \\
\text{plus}(M, s(z), s(s(z))) & \quad \text{(plus-s) and } M = s(M_1)
\end{align*}
\]
BACKTRACKING

- Backtrack and try (plus-z):

\[
\frac{\text{plus}(M_1, s(z), s(z))}{\text{plus}(M, s(z), s(s(z)))}
\]

Success!

(plus-z) and \( M_1 = z \)
(plus-s) and \( M = s(M_1) \)
**Subgoal Order**

- When a rule contains multiple premises, we need to determine which premise (or subgoal) to attempt first.

\[
J_1 \ldots J_k
\]

- The order of subgoal evaluation has a significant impact on the computation:
  - Complete in a few steps in some order
  - Non-terminating in other order
  - Corresponds to order of search (DFS, BFS, etc.)
**Horn Clauses**

- An inference rule can be represented as a *horn clause*.
- A Horn clause has a head $h$, which is a *predicate*, and a body, which is a list of *predicates* $p_1, p_2, \ldots, p_n$.
- It is written as:
  
  $$ h \leftarrow p_1, p_2, \ldots, p_n $$

- This means, “$h$ is true only if $p_1, p_2, \ldots, p_n$ are simultaneously true.”
- E.g., the Horn clause:

  $$ \text{snowing}(C) \leftarrow \text{precipitation}(C), \text{freezing}(C) $$

  says, “it is snowing in city $C$ only if there is precipitation in city $C$ and it is freezing in city $C$.”
HORN CLAUSES AND PREDICATES

- Any Horn clause
  \[ h \leftarrow p_1, p_2, \ldots, p_n \]
  can be written as a predicate:
  \[ p_1 \land p_2 \land \ldots \land p_n \supseteq h \]
  or equivalently:
  \[ \neg(p_1 \land p_2 \land \ldots \land p_n) \lor h \]
- But not every predicate can be written as a Horn clause.
- E.g., \textit{literate}(x) \supseteq \textit{reads}(x) \lor \textit{writes}(x)
Resolution and Unification

If $h$ is the head of a Horn clause

$$h \leftarrow \text{terms}$$

and it matches one of the terms of another Horn clause:

$$t \leftarrow t_1, h, t_2$$

then that term can be replaced by $h$’s terms to form:

$$t \leftarrow t_1, \text{terms}, t_2$$

During resolution, assignment of variables to values is called instantiation.

Unification is a pattern-matching process that determines what particular instantiations can be made to variables during a series of resolutions.
Example

- The two clauses:
  
  $\text{speaks}(\text{Mary, English})$
  
  $\text{talkswith}(X, Y) \leftarrow \text{speaks}(X, L), \text{speaks}(Y, L), X \neq Y$

  can resolve to:
  
  $\text{talkswith}(\text{Mary, Y}) \leftarrow$
  
  $\text{speaks}(\text{Mary, English}),$
  
  $\text{speaks}(Y, \text{English}), \text{Mary} \neq Y$

- The assignment of values $\text{Mary}$ and $\text{English}$ to the variables $X$ and $L$ is an instantiation for which this resolution can be made.
In logic programming the program declares the goals of the computation, not the method for achieving them.

Logic programming has applications in AI and databases.
- Natural language processing (NLP)
- Automated reasoning and theorem proving
- Expert systems (e.g., MYCIN)
- Database searching, as in SQL (Structured Query Language)

Prolog emerged in the 1970s. Distinguishing features:
- Nondeterminism
- Backtracking
Prolog programs are made from *terms*, which can be:

- Variables
- Constants
- Structures

*Variables* begin with a capital letter, like *Bob*.

*Constants* are either integers, like *24*, or atoms, like *the*, *zebra*, *‘Bob’*, and *‘.’*.

*Structures* are predicates with arguments, like:

- `n(zebra)`, `speaks(Y, English)`, and `np(X, Y)`

The arity of a structure is its number of arguments (1, 2, and 2 for these examples).
Facts, Rules, and Programs

- A Prolog fact is a Horn clause without a right-hand side. Its form is (note the required period .):

  term.

- A Prolog rule is a Horn clause with a right-hand side. Its form is (note :- represents ← and the required period .):

  term :- term₁, term₂, ... termₙ.

- A Prolog program is a collection of facts and rules.
**Example Program**

speaks(allen, russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(X, Y) :- speaks(X, L), speaks(Y, L), X \(\neq\) Y.

- This program has four facts and one rule.
- The rule *succeeds* for any instantiation of its variables in which all the terms on the right of :- are simultaneously true. E.g., this rule succeeds for the instantiation \(X=\text{allen}, Y=\text{mary},\) and \(L=\text{russian}\.\)
- For other instantiations, like \(X=\text{allen}\) and \(Y=\text{bob}\), the rule *fails*.
Searching for Success: Queries

- A *query* is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest. E.g.,
  
  ?- speaks(Who, russian).

  asks for an instantiation of the variable *Who* for which the query *speak*(*Who*, russian) succeeds.

- A program is loaded by the query *consult*, whose argument names the program. E.g.,
  
  ?- consult(speaks).

  loads the program named *speaks*, given on the previous slide.

- The word “consult” is reminiscent of the “expert system”.

ANSWERING THE QUERY: UNIFICATION

- To answer the query:
  \[- \text{speaks}(\text{Who, russian}).\]

- Prolog considers every fact and rule whose head is \text{speaks}. (If more than one, consider them in \textit{some} order – non-deterministic!)

- Resolution and unification locate all the successes:
  \[
  \text{Who} = \text{allen} ; \\
  \text{Who} = \text{mary} ; \\
  \text{No}
  \]

  Each semicolon (;) asks, “Show me the next success.”
**Search Trees**

- First attempt to satisfy the query `- talkswith(Who, allen).`
DATABASE SEARCH - THE FAMILY TREE
**PROLOG PROGRAM**

mother(mary, sue).  father(john, sue).
mother(mary, bill).  father(john, bill).
mother(sue, nancy).  father(bob, nancy).
mother(sue, jeff).   father(bob, jeff).
mother(jane, ron).   father(bill, ron).

parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).

grandparent(C,D) :- parent(C,E), parent(E,D).
**Some Database Queries**

- Who are the parents of jeff?
  
  ```
  ?- parent(Who, jeff).
  Who = bob;
  Who = sue
  ```

- Find all the grandparents of Ron.
  
  ```
  ?- grandparent(Who, ron).
  ```

- What about siblings? Those are the pairs who have the same parents.
  
  ```
  ?- sibling(X, Y) :- parent(W, X), parent(W, Y), X\=Y.
  ```
LISTS

- A *list* is a series of terms separated by commas and enclosed in brackets.
  - The empty list is written `[]`.
  - The sentence “The giraffe dreams” can be written as a list: `[the, giraffe, dreams]`
  - A “don’t care” entry is signified by `_`, as in `[_, X, Y]`
  - A list can also be written in the form: `[Head | Tail]`
  - The functions `append` joins two lists, and `member` tests for list membership.
APPEND FUNCTION

append([], X, X).
append([Head | Tail], Y, [Head | Z]) :-
    append(Tail, Y, Z).

- This definition says:
  1. Appending any list (X) to the empty list returns an unchanged list (X again).
  2. If Y is appended to Tail to get Z, then Y can be appended to a list one element larger [Head | Tail] to get [Head | Z].

- Note: The last parameter designates the result of the function. So a variable must be passed as an argument.
MEMBER FUNCTION

member(X, [X | _]).
member(X, [__ | Y]) :- member(X, Y).

- The test for membership succeeds if either:
  1. X is the head of the list [X | __]
  2. X is not the head of the list [__ | Y], but X is a member of the list Y.

- Notes: pattern matching governs tests for equality.
- Don’t care entries (__) mark parts of a list that aren’t important to the rule.
More List Functions

- X is a prefix of Z if there is a list Y that can be appended to X to make Z. That is:
  \[ \text{prefix}(X, Z) :\text{-} \text{append}(X, Y, Z) \].

- Similarly, Y is a suffix of Z if there is a list X to which Y can be appended to make Z. That is:
  \[ \text{suffix}(Y, Z) :\text{-} \text{append}(X, Y, Z) \].

- So finding all the prefixes (suffixes) of a list is easy. E.g.:
  
  \[\text{-} \text{prefix}(X, [\text{my, dog, has, fleas}]).\]
  
  \[
  X = []; \\
  X = [\text{my}]; \\
  X = [\text{my, dog}]; \\
  \ldots
  \]
PRACTICAL ASPECTS OF PROLOG

- Tracing
- The Cut
- Negation
- The is, not, and Other Operators
- The Assert Function
TRACING

To see the dynamics of a function call, the `trace` function can be used. E.g., if we want to trace a call to the following function:

```
factorial(0, 1).
factorial(N, Result) :- N > 0, M is N - 1,
                   factorial(M, SubRes), Result is N * SubRes.
```

we can activate `trace` and then call the function:

```
?- trace(factorial/2).
?- factorial(4, X).
```

*Note*: the argument to `trace` must include the function’s arity.
TRACING OUTPUT

?- factorial(4, X).
Call: ( 7) factorial(4, _G173)
Call: ( 8) factorial(3, _L131)
Call: ( 9) factorial(2, _L144)
Call: (10) factorial(1, _L157)
Call: (11) factorial(0, _L170)
Exit: (11) factorial(0, 1)
Exit: (10) factorial(1, 1)
Exit: ( 9) factorial(2, 2)
Exit: ( 8) factorial(3, 6)
Exit: ( 7) factorial(4, 24)

X = 24

These are temporary variables

These are levels in the search tree
The Cut

- The cut is an operator (!) inserted on the right-hand side of a rule.
- **semantics**: the cut forces the subgoals to its left not to be retried if the right-hand side succeeds once, i.e. no backtrack to the left of (!).
- E.g (bubble sort):
  
  \[
  \text{bsort}(L, S) :- \text{append}(U, [A, B | V], L), \\
  \quad B < A, !, \\
  \quad \text{append}(U, [B, A | V], M), \\
  \quad \text{bsort}(M, S). \\
  \quad \text{bsort}(L, L).
  \]

- So this code gives one answer rather than many.
- Limit search space and improves performance.
BUBBLE SORT TRACE

?- bsort([5,2,3,1], Ans).
Call: ( 7) bsort([5, 2, 3, 1], _G221)
Call: ( 8) bsort([2, 5, 3, 1], _G221)
...
Call: (12) bsort([1, 2, 3, 5], _G221)
Redo: (12) bsort([1, 2, 3, 5], _G221)
...
Exit: ( 7) bsort([5, 2, 3, 1], [1, 2, 3, 5])

Ans = [1, 2, 3, 5] ;

No

Without the cut, this would have given some wrong answers.
**THE IS OPERATOR**

- *is* instantiates a temporary variable. E.g., in

  factorial(0, 1).

  factorial(N, Result) :- N > 0, M is N - 1,
  factorial(M, SubRes), Result is N * SubRes.

- Here, the variables *M* and *Result* are instantiated
  This is like an assignment to a local variable in C-like languages.

- You could have said:
  factorial(N, N * SubRes) :- N > 0, factorial(N - 1, SubRes).
**OTHER OPERATORS**

- Prolog provides the operators
  
  \[
  +, -, *, /, ^, =, <, >, \geq, \leq, =\n  \]

  with their usual interpretations.

- The **not** operator is implemented as goal failure. E.g.,

  ```prolog
  factorial(N, 1) :- N < 1.
  factorial(N, Result) :- not(N < 1), M is N - 1,
  factorial(M, P),
  Result is N * P.
  ```

  is equivalent to using the cut (!!) in the first rule.
THE ASSERT FUNCTION

- The assert function can update the facts and rules of a program dynamically. E.g., if we add the following to the foregoing database program:
  
  ```
  assert(mother(jane, joe)).
  ```

- Then the query:
  
  ```
  mother(jane, X).
  ```

- gives:
  
  ```
  X = ron;
  X = joe;
  No
  ```
**Prolog Examples**

1. Solving Word Puzzles
   Nondeterminism seeks all solutions, not just one

2. Natural Language Processing
   One of Prolog’s traditional research applications
A simple example:
Baker, Cooper, Fletcher, Miller, and Smith live in a five-story building. Baker doesn't live on the 5th floor and Cooper doesn't live on the first. Fletcher doesn't live on the top or the bottom floor, and he is not on a floor adjacent to Smith or Cooper. Miller lives on some floor above Cooper. Who lives on what floors?

We can set up the solution as a list of five entries:
[floor(_, 5), floor(_, 4), floor(_, 3), floor(_, 2), floor(_, 1)]

The don’t care entries are placeholders for the five names.
MODELING THE SOLUTION

- We can identify the variables B, C, F, M, and S with the five persons, and the structure floors(Floors) as a function whose argument is the list to be solved.

- Here’s the first constraint:
  \[
  \text{member(floor(baker, B), Floors), B\leq5}
  \]
  which says that *Baker doesn't live on the 5th floor.*

- The other four constraints are coded similarly, leading to the following program:
**PROLOG SOLUTION**

```prolog
floors([floor(_,5),floor(_,4),floor(_,3),floor(_,2),
        floor(_,1)]).

building(Floors) :- floors(Floors),
    member(floor(baker, B), Floors), B \= 5,
    member(floor(cooper, C), Floors), C \= 1,
    member(floor(fletcher, F), Floors), F \= 1, F \= 5,
    member(floor(miller, M), Floors), M > C,
    member(floor(smith, S), Floors), not(adjacent(S, F)),
    not(adjacent(F, C)),
    print_floors(Floors).
```
Auxiliary Functions

- Floor adjacency:
  \[ \text{adjacent}(X, Y) :\ X =:= Y+1. \]
  \[ \text{adjacent}(X, Y) :\ X =:= Y-1. \]
  \textit{Note:} \( =:= \) tests for numerical equality.

Displaying the results:

\[ \text{print_floors}([A \mid B]) :\ write(A), \ \text{nl}, \ \text{print_floors}(B). \]
\[ \text{print_floors}([]). \]
\textit{Note:} \texttt{write} is a Prolog function and \texttt{nl} stands for “new line.”

- Solving the puzzle is done with the query:
  \[ ?- \text{building}(X). \]
which finds an instantiation for \( X \) that satisfies all the constraints.
BNF can define natural language (e.g., English) syntax.
- This was the original purpose of BNF when it was invented by Chomsky in 1957.

A Prolog program can model a BNF grammar.
- This was an original purpose of Prolog when it was designed in the 1970s.

A Prolog list can model a sentence.
- E.g., [the, giraffe, dreams]

So running the program can parse a sentence.
NLP Example

Consider the following BNF grammar and parse tree:

\[
\begin{align*}
  s & \rightarrow np \ vp \\
np & \rightarrow det \ n \\
vp & \rightarrow tv \ np \\
      & \quad \rightarrow iv \\
det & \rightarrow the \\
n & \rightarrow giraffe \\
      & \quad \rightarrow apple \\
iv & \rightarrow dreams \\
tv & \rightarrow eats
\end{align*}
\]

Here, \(s\), \(np\), \(vp\), \(det\), \(n\), \(iv\), and \(tv\) denote “sentence,” “noun phrase,” “verb phrase,” “determiner,” “noun,” “intransitive verb,” and “transitive verb.”
PROLOG ENCODING (NAÏVE VERSION)

s(X, Y) :- np(X, U), vp(U, Y).
np(X, Y) :- det(X, U), n(U, Y).
vp(X, Y) :- iv(X, Y).
vp(X, Y) :- tv(X, U), np(U, Y).
det([the | Y], Y).
n([giraffe | Y], Y).
n([apple | Y], Y).
iv([dreams | Y], Y).
tv([eats | Y], Y).

The first rule reads, “list X is a sentence leaving tail Y if X is a noun phrase leaving tail U and U is a verb phrase leaving tail Y.”
**Example Trace**

The query asks, “can you resolve [the, giraffe, dreams] as an s, leaving tail []?”

Yes

The result is success.
Definite Clause Grammars (DCGs)

s --> np, vp.
np --> det, n.
vp --> iv.
vp --> tv, np.
det --> [the].
n --> [giraffe].
n --> [apple].
iv --> [dreams].
tv --> [eats].

This replaces
s(X, Y) :- np(X, U), vp(U, V).
but it means the same thing.

Note: This form looks more like a series of BNF rules, and it’s simple to write!
Generating Parse Trees

- Terms and variables can be added to the DCG rules so that a parse tree can be generated from a query.
- E.g., if we change
  \[ s \rightarrow np, \text{vp}. \]
to
  \[ s(s(NP, VP)) \rightarrow np(NP), \text{vp}(VP). \]
- The variables NP and VP can capture intermediate subtrees.

- When all rules are augmented in this way, the query
  \[ ?- s(Tree, [the, giraffe, dreams], []). \]
delivers the parse tree as a parenthesized list:
  \[ Tree = s(np(det(the), n(giraffe)), vp(iv(dreams))) \]