Announcements

- All homework solutions are available.

- **Final exam: Tuesday, December 18, 4:00pm - 7:00pm, SEC 212**

- Project 3 extension needed?

- Friday, December 14, 10:30am - noon, review session, location will be announced later. Please see our class website.

- Project 1: Ten submissions do not run correctly (syntax errors, ...). Grader (Manish) will be available after review session if there are “minor” fixes necessary to make your code run.
Project 3 – QuickShift with Tradeoff Analysis

Input Image “flowers2.pnm”

Output Image “flowers2-cpu.pnm”
Predicate Logic – first order logic

Syntax — Well formed formulae (wff):

- **term**: constant symbols, variables, and n-place function symbols followed by n terms enclosed in parenthesis

- **atomic formula**: n-place predicate symbols followed by n terms enclosed in parenthesis

- **wff**: atomic formulae and expressions of the form $(\alpha \rightarrow \beta)$, $(\alpha \lor \beta)$, $(\alpha \land \beta)$, $(\neg \alpha)$, $\forall x \alpha$, or $\exists x \alpha$, where $\alpha$ and $\beta$ are wffs and $x$ is a variable

Examples:

\[
\forall x(p(x) \rightarrow q(x))
\]

\[
\forall x \exists y(p(x) \lor q(f(y))
\]

A wff is closed if it does not contain any free variables. To simplify the discussion, we will assume that all wffs are closed.
Predicate Logic – first order logic

Semantics — Interpretations:
An interpretation $I$ consists of

- a non-empty set $D$, called the universe
- a mapping that assigns to each constant $c$ a fixed element $c^I \in D$
- a mapping that assigns to each n-place function symbol $f$ an n-ary function $f^I : D^n \rightarrow D$
- a mapping that assigns to each n-place predicate symbol $p$ an n-ary predicate $p^I : D^n \rightarrow \{\text{true, false}\}$

Note:
- logic connectives $\rightarrow, \lor, \land, \neg$ have their usual propositional meaning
- $\forall x$ and $\exists x$ mean “for all $x$ in $D$” and “there exists an $x$ in $D$”, respectively
Predicate Logic – first order logic

An interpretation $I$ satisfies wff $\alpha$ ($|=_I \alpha$) iff $\alpha$ evaluates to true under the interpretation $I$. Such an $I$ is called a model of $\alpha$.

A wff $\alpha$ is valid ($|= \alpha$) iff it is satisfied for all of its interpretations.

A wff $\alpha$ is unsatisfiable iff it is not satisfiable in any of its interpretations.

A set $\Gamma$ of wffs logically implies a wff $\alpha$ ($\Gamma |= \alpha$) iff for every interpretation that satisfies all members in $\Gamma$, $\alpha$ is also satisfied.

**Question:** Is there a mechanical way to derive all wffs that are logically implied by a set of wffs $\Gamma$?
A deductive calculus consists of

- A set $\Delta$ of wffs, called logical axioms
- A set of inference rules

A wff $\alpha$ is a theorem of a set $\Gamma$ of wffs ($\Gamma \vdash \alpha$) iff $\alpha$ belongs to the set of formulae that can be generated from $\Gamma \cup \Delta$ by a finite sequence of inference rule applications. Such a sequence is called a deduction.

Without going into details about its infinite set $\Delta$, there is a deductive calculus that uses only a single inference rule, called modus ponens:

\[
\begin{array}{c}
\alpha, \alpha \to \beta \\
\hline 
\beta
\end{array}
\]

Example:
Assume that $\Delta$ contains $\forall x p(x) \to p(a)$. Then we can proof

\[
\{ \forall x p(x), (p(a) \to q(a)) \} \vdash q(a)
\]

with two applications of the inference rule.
Two major properties of a deductive calculus:

- A deductive calculus is **sound** iff
  \[ \Gamma \vdash \alpha \text{ implies } \Gamma \models \alpha \]

- A deductive calculus is **complete** iff
  \[ \Gamma \models \alpha \text{ implies } \Gamma \vdash \alpha \]

In other words, *soundness* means “whatever can be proven is valid”, and *completeness* means “whatever is valid can be proven”. A calculus that is not sound is pretty useless. A calculus that is sound and complete is the best we can do.

Gödel showed in 1930 that first order logic is complete, i.e., there is a sound deductive calculus for first order logic that is complete.

Questions:

- Is propositional logic decidable?
- Is \( \Gamma \vdash \alpha \) decidable?
- Is second order logic (allows quantification over predicates) complete?
Logic Programming and Prolog

Logic programming languages are not procedural or functional (Scott Chap. 12 - Logic Languages).

- Specify *relations* between objects
  - `larger(3,2)`
  - `father(tom,jane)`

- Separate logic from control:
  - Programmer declares **what** facts and relations are true
  - System determines **how** to use facts to solve problems

- Based on Predicate Logic (first order logic)

- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
Free and bound variables

A variable $x$ can occur in a wff

- within a term (use), or
- next to a quantifier $\forall$ or $\exists$ (definition).

Any use of a variable is bound to the “closest” surrounding definition of the variable, if such a definition exists.

For the wffs $\forall x \alpha$ or $\exists x \alpha$, $\alpha$ is called the scope of definition $x$.

Example:

$$(\forall x (p(x) \rightarrow \exists y \exists x (q(x) \vee q(y))) \vee x)$$

Question: What are the bindings between the uses and definitions of variables? Note: the same variable can be defined or used several times in a wff.

The use of a variable is free in a wff if there is no matching definition. Otherwise it is bound.
Prolog and predicate logic

Prolog is based on *Horn clauses* (clauses). Formulae have the form:

\[ \forall X ( (\text{male}(X) \land \text{parent}(X, \text{jane})) \rightarrow \text{father}(X, \text{jane}) ) \]

written in Prolog as

\[ \text{father}(X, \text{jane}) : - \text{male}(X), \text{parent}(X, \text{jane}). \]

In general,

\[ A : - B_1, B_2, \ldots, B_n. \]

where \( A, B_1, \ldots, B_n \) are atomic formulae and \( n \geq 0 \). \( A \) is called the *conclusion* and the \( B_i \)'s are called *subgoals* or *conditions*.

Variables can occur in the atomic formulae. All variables are implicitly \( \forall \)-quantified and therefore there are no free variables. Note: Scope of variables is restricted to a single clause.
Prolog

\[ A : - B_1, B_2, \ldots, B_n. \]

Three different instantiations:

- **rule** — conclusion (head) and conditions (tail) are non-empty
  
- **facts** or **assertions** — conditions are empty.
  
  Example: father(tom, jane).

- **query** or **goal** — conclusion is empty.
  
  Example: ?- father(X, jane).

In Prolog:

- All variables are capitalized
- All constants are in lower case
- All predicates are in lower case
Use **prolog** command (SWI-Prolog) on ilab cluster to start prolog interpreter. Exit using **CntrlD**. A Prolog program starts with declarations of the basic facts.

```
male(albert). \---------- a fact
female(alice).   Facts are put in file "family.pl"
male(edinward).
female(victoria).
parent(albert,edward).
pARENT(victoria,edward).
pARENT(albert,alice).
pARENT(victoria,alice).
```

```bash
> prolog
| ?- [family]. \------------ loads file
| yes
| ?- male(albert). \------------ a query
| yes
| ?- male(alice).
| no
| ?- parent(albert,edward).
| yes
| ?- parent(bullwinkle,edward).
| no
```

Limited use: need variables and deductive rules.
Variables and Unification

| ?- female(X).  
X = alice  
| ?- female(X).  
X = alice ; <------------- ‘;’ means look further  
X = victoria ;  
no

X is **unified** to all possible values that make the query female(X) true.

| ?- parent(P,edward).  
P = albert ;  
P = victoria ;  
no

P is unified to all possible values that make the query parent(P,edward) true.

⇒ search with pattern matching
Prolog Horn Clause Examples

A *Horn clause with no tail*:

\[ \text{male(albert)}. \]

I.e., a fact: albert is a male dependent on no other conditions

A *Horn clause with a tail*:

\[ \text{father(albert,edward):-} \\
\quad \text{male(albert), parent(albert,edward)}. \]

I.e., a rule: albert is the father of edward if albert is male and albert is a parent of edward’s.
Horn Clauses with Variables

Variables may appear in the head and tail of a Horn clause:

- \( c(X_1, \ldots, X_n) \leftarrow h(X_1, \ldots, X_n) \).
  
  “For all values of \( X_1, \ldots, X_n \), the formula \( c(X_1, \ldots, X_n) \) is true if the formula \( h(X_1, \ldots, X_n) \) is true”

- \( c(X_1, \ldots, X_n) \leftarrow h(X_1, \ldots, X_n, Y_1, \ldots, Y_k) \).
  
  “For all values of \( X_1, \ldots, X_n \), the formula \( c(X_1, \ldots, X_n) \) is true if there exist values of \( Y_1, \ldots, Y_k \) such that the formula \( h(X_1, \ldots, X_n, Y_1, \ldots, Y_k) \) is true”

Example from logic:

\[
\forall X (p(a) \leftarrow q(X))
\]

is logically equivalent to

\[
(p(a) \leftarrow \exists X q(X))
\]
Examples

father(X,Y):- male(X), parent(X,Y).
(X is the father of Y if X is male and X is a parent of Y)

?- father(F,edward).
F = albert ;
no

child_of(C,P):- parent(P,C).

?- child_of(C,P).
C = edward
P = albert ;
C = edward
P = victoria ;
C = alice
P = albert ;
C = alice
P = victoria ;
no
Examples

sibling(X,Y):- parent(P,X), parent(P,Y).

|  ?- sibling(alice,A).
A = edward ;
A = alice ;
A = edward ;
A = alice ;
no
Rule Ordering and Unification

1. rule ordering used in search

2. unification requires two instances of the same variable to get the same value

3. unification does not require differently named variables to get different values: hence, sibling(Edward, Edward).

4. all rules searched if requested by ‘;’
Prolog and predicate logic

Horn clauses allow efficient implementation of “backward” deductions through backward chaining.

Predicate logic is declarative. Semantics is independent of HOW to prove theorems. There is no control information other than implied by logical inference. Control is an efficiency issue, not a correctness issue.

Prolog has a **procedural** interpretation with a **declarative** flavor. Goal: Separate control component of program (HOW) from description of desired outcome (WHAT).

```
algorithm = logic + control
```

Prolog: Control is part of the semantics

Things we will talk about:

- subgoal and rule selection
- cuts
- negation as failure
Prolog Search Tree - represents backchaining

male(albert).
female(alice).
male(Edward).
female(victoria).
parent(albert, edward).
parent(victoria, edward).
parent(albert, alice).
parent(victoria, alice).
sibling(X, Y) :- parent(P, X), parent(P, Y).

Prolog Search Tree
Control in Prolog

- **goal rule** — choose the leftmost subgoal
- **rule order** — select the first applicable rule

Prolog uses a depth-first search of the tree. This can be “dangerous” if the tree has infinite subtrees. Replacement of a goal by the subgoals of right-hand side of rule is called *backward chaining*.
A procedural interpretation

\[ A : \neg B_1, B_2, \ldots, B_n. \]

- Interpret each clause as a procedure definition: conclusion is procedure name or head, and conditions are the body.
- Main program is procedure body with no name (original query).
- To execute the body of a procedure \( A \), call each procedure \( B_1, B_2, \ldots, B_n \).
- Procedures are invoked by unification, a generalized pattern match. Find the most general unifier (mgu) of the selected call (subgoal) and the head of the selected clause (rule).
- Execution terminates when all procedures have successfully terminated.
- A procedure with no body terminates as soon as the unifying substitution is made.
Quick Review

- Prolog has a **procedural** interpretation with a **declarative** flavor.

- Control in Prolog
  - **goal rule** — choose the leftmost subgoal
  - **rule order** — select the first applicable rule

- Replacement of a goal by the subgoals of right-hand side of rule is called **backward chaining**.

- Prolog uses a depth-first search over the **Prolog Search Tree**. This can lead to non-termination if the tree has infinite subtrees.

- A **unifier** is a substitution of variables by terms such that, when applied to both the call and the head of the selected rule, makes them syntactically identical. The **most general unifier** (mgu) makes as few and as general bindings as possible.

- Computation builds up a composition of unifiers. If program terminates successfully, the bindings of the variables in the initial query are the output of the program.
Prolog: unification

A unifier is a substitution of variables by terms such that, when applied to both the call and the head of the selected rule, makes them syntactically identical. The most general unifier (mgu) makes as few and as general bindings as possible.

Note: “=” is the unification operator in Prolog

Example:

times(Z, times(Y, 7) ) = times(4, W)

$\Theta_1 =$
\{ $Z \rightarrow 4, Y \rightarrow plus(3, 5), W \rightarrow times(plus(3, 5), 7)$\}

$\Theta_2 =$
\{ $Z \rightarrow 4, W \rightarrow times(Y, 7)$\}

Question: Which of the two substitutions is more general?

Note: mgu $\theta$ is unique modulo renaming of variables
Unification — formal definition

• A substitution $\sigma$ is a mapping from variables to terms.

  The result of the substitution $T\sigma$ is recursively defined ("=" means textually identical):

  $- X\sigma = U$ if $X \rightarrow U$ is in $\sigma$
  $- X\sigma = X$ otherwise
  $- (f(T_1, T_2))\sigma = f(U_1, U_2)$ if $T_1\sigma = U_1$ and $T_2\sigma = U_2$

  Assumption: if $\sigma$ maps $X$ to $T$, then $X$ does not occur in $T$ (occurs check → will discuss later).

• A term $U$ is an instance of $T$ if $U = T\sigma$ for some substitution $\sigma$.

• Two terms $T_1$ and $T_2$ unify if $T_1\sigma = T_2\sigma$ for some substitution $\sigma$. $\sigma$ is called a unifier of $T_1$ and $T_2$.

• The substitution $\theta$ is the most general unifier (mgu) of $T_1$ and $T_2$ if $\forall$ other unifiers $\sigma'$, $T_1\sigma'$ is an instance of $T_1\theta$. 
Example list of subgoals:

\[- A_1, \ldots, A_i, \ldots A_m. \quad m \geq 0\]

Invoke procedure \(A_i\) with rule \(A : - B_1, \ldots B_n.\)

Unify \(A_i\) with \(A\), i.e., \(A_i \Theta = A \Theta\). \(\Theta\) maps variables to terms. The resulting list of subgoals is:

\[- (A_1, \ldots, A_{i-1}, B_1, \ldots B_n, A_{i+1} \ldots A_m) \Theta.\]

Note: Unification performs the basic data manipulation operations of

- assignment
- parameter passing
- data selection
- data construction

Note:
Computation builds up a composition of unifiers. If program terminates successfully, the bindings of the variables in the initial query are the output of the program.
Prolog: backward-chaining computation

1. start with initial query
2. Select a call to execute
3. Select a procedure to use in executing the chosen call
4. Standardize — rename variables to ensure that there are no variables that occur both in the current set of calls and in the selected procedure (renaming)
5. Find \textit{mgu} of the selected call and the name of the selected procedure
6. Replace the selected call by the body of the procedure
7. Apply the \textit{mgu} to the new set of calls resulting from previous step
8. If no calls remain, terminate successfully; If no procedure name found to match the selected call, back up and redo the previous call, using a different procedure than the one already used; if all backtrack options are exhausted, terminate with failure
Order of Arguments, Rules, and Subgoals

| ?- sibling(A,alice).
A = edward ;
A = edward ;
A = alice ;
A = alice ;
no

Note: arguments are interchangeable, but ordering affects order of search.

In General:
Order of subgoals and rules may make a difference in terms of efficiency.

- Order may determine whether resolution terminates or not.

- You can write inefficient logic. Example:
  sort(x,y) :- permut(x,y), ordered(y)
Prolog syntax (subset) in EBNF

\[\begin{align*}
<\text{clause}> & ::= <\text{atmf}> . \mid \\
<\text{atmf}> & ::= <\text{atmfs}> . \mid \\
? & ::= <\text{atmfs}> . \\
<\text{atmfs}> & ::= <\text{atmf}> \{, <\text{atmf}> \} \\
<\text{atmf}> & ::= <\text{pred}>(<\text{terms}> ) \\
<\text{terms}> & ::= <\text{term}> \{, <\text{term}> \} \\
<\text{term}> & ::= <\text{variable}> \mid <\text{constant}> \mid <\text{func}>(<\text{terms}> ) \mid <\text{list}> \\
<\text{variable}> & ::= <\text{uppercase}\_\text{word}> \\
<\text{constant}> & ::= <\text{lowercase}\_\text{word}> \mid \text{number} \\
<\text{pred}> & ::= <\text{lowercase}\_\text{word}> \\
<\text{func}> & ::= <\text{lowercase}\_\text{word}> \\
<\text{list}> & ::= [ ] \mid [ <\text{head}> \mid <\text{tail}> ] \\
<\text{head}> & ::= <\text{term}> \\
<\text{tail}> & ::= <\text{list}> \\
\end{align*}\]
Lists in Prolog

[a, b] is an abbreviation for [a | [b | []]]

<table>
<thead>
<tr>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b, c]</td>
<td>a</td>
</tr>
<tr>
<td>[X, [foo], Y]</td>
<td>X</td>
</tr>
<tr>
<td>[a, [b, c], d]</td>
<td>a</td>
</tr>
</tbody>
</table>

[a, b, c]

[1 | 2]

[a, [b, c], d]

[1, 2]
More examples: Lists in Prolog

Can you unify the following lists?

\[ X, \text{abc}, Y \] and \[ X, \text{abc} | Y \] ?

\[ \text{a, b | Z} \] and \[ X | Y \] ?

\[ \text{a, b, c} \] and \[ X | Y \] ?
Operations on Lists in Prolog

member(A, [A | B]).
member(A, [B | C]) :- member(A, C).

?- member(a, [a,b]).

?- member(a, [b,c]).

?- member(X, [a,b,c]).

?- member(a, [b,c,X]).

?- X = [1,2,3], member(a,X).

?- member(a,X), X = [1,2,3].
member Query: Prolog Search Tree

member(A, [A | B]).
member(A, [B | C]) :- member(A, C).

?- X = [1,2,3], member(a,X).
member – Using Placeholders

Another way or writing the member predicate:

\[
\text{member}(A, [A | \_]).
\]
\[
\text{member}(A, [\_ | C]) :– \text{member}(A, C).
\]

\[?- \text{member}(a,X), X = [1,2,3].\]

Leads to infinite computation.

\[X = [a|\_] ; \quad \text{\(a\) can be the first element of \(X\)}\]
\[X = [\_, a|\_] ; \quad \text{\(a\) can be the second element of \(X\)}\]
\[X = [\_, \_, a|\_] ; \quad \text{\(a\) can be the third element of \(X\)}\]

“\_” is a special Prolog variable; a placeholder (fresh variable) is generated by the Prolog system for each occurrence of “\_”. 
Arithmetic in Prolog

?- AGE is 1995 - 1956.
AGE = 39 ;
no

AGE = 1995-1956;
no

no

?- 39 is DATE - 1956.
! Instantiation error in argument 2 of is/2
! goal: 39 is _6668-1956

At the time Prolog begins processing a goal of the form: X is <Exp>, <Exp> must be a fully instantiated arithmetic expression, i.e., may not contain any variables after unification.

⇒ Arithmetic programs are not always invertible.

?- AGE is DATE - 1956, DATE is 1995.
?- DATE is 1995, AGE is DATE - 1956.
Arithmetic in Prolog: Examples

factorial(0,1).
factorial(X,Y) :- W is X-1,  
                   factorial(W,Z),  
                   Y is Z*X.

This calculates X! if X is bound to an integer. Otherwise it aborts in the first “is” clause.

?- factorial(3,6).
yes

?- factorial(5,Z).
Z = 120

?- factorial(Y,6).
! Instantiation error in argument 2 of is/2
! goal: _6568 is _6571-1
A cut (denoted !) prunes parts of a Prolog search tree by restricting backtracking.

Cuts are control features, i.e., make Prolog depart even further from “declarative logic”.

Cut can occur on right-hand side of rules:
\[ B : - C_1, \ldots C_{j-1}, !, C_{j+1}, \ldots C_k \]

Backtrack past \( C_{j-1}, \ldots C_1, B \) without considering any remaining, alternative rules for them.

The cut is a subgoal that always succeeds.

Optimize time and space of the computation.

Green cuts: Prune parts of search tree that cannot contain any successful solutions.

Red cuts: All others
Examples:

\[ \text{conclusion}(S) :- \text{guess}(S), \!, \text{verify}(S) \]

Eliminates all but the first successful guess

Note: if \text{guess}(S) fails, another rule for \text{conclusion}(S) can be selected.
Negation as failure

not(X) :- X, !, fail.
not(_).

If \textbf{X succeeds} in first rule, then the rule is forced to fail by the last term. We cannot backtrack over the cut in the first rule and the cut prevents us from accessing the second rule.

If \textbf{X fails} in first rule, then the second rule succeeds, since \_ unifies with anything.

Some interesting behavior:

?- X = 2, not(X = 1).
X=2
?
?- not(X = 1), X = 2
no

Note

- The implications of negation as failure are very complex. Be careful.
Prolog and Predicate Logic: Review

- Horn clause logic is a proper subset of predicate logic in terms of syntax and semantics. The following is not a Prolog program:

\[
Q(a) \lor Q(b).
\]

\[
?- Q(X).
\]

In predicate logic, the answer should be YES, since \(Q(a) \lor Q(b) \models \exists x Q(x)\). However, we don’t know what the substitution for \(X\) should be.

- In Horn clause logic, we can always compute definite answers through resolution and unification:

If \(P\) is a Prolog program, then

\[
P \models \exists Q(x) \implies \text{there is a unifier } \theta \text{ such that } P \models Q(x)\theta.
\]

This means that we actually can compute answers, i.e., substitutions for variables in the input query. For the example above, we don’t know whether \(\theta = \{X \rightarrow a\}\) or \(\theta = \{X \rightarrow b\}\) is a correct answer.
That’s it!

Reminder: Review session on Friday at 10:30am

Final exam: Tuesday, December 18, 4:00-7:00pm, SEC 212