Announcements

- Exam 19 dec moved to 15:15-18:00.
- Guest lecture moved to 14 dec.
- Deadlines reports: 1 December
- Student presentations in 2 sessions: 30 nov and 7 december

Logic programming

- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

The idea of logic programming

- A program is a collection of facts and rules (Horn clauses)
- The user can ask questions to the language system by giving a goal to prove. The language system tries to prove the goal by using the clauses given in the program.
- Goal - oriented

Logical programming

Facts about Prolog

- Prolog = Programming in Logic (University of Marseille, mid 70s)
- The most important logic language (other - Godel, Escher)
- A declarative language. Programmer only specifies the problem and the language system finds a solution.
- Used in AI applications (automated reasoning systems, expert systems)
**Terms**

- All Prolog programs and data are built from terms:
  - Constants: 1, 1.23, fred, *, =, []
  - Variables: X, Y, Fred, Child, _
  - Compound terms: parent(X, Y)

**The Prolog Database**

- A Prolog language system maintains a database of facts and rules of inference
- A Prolog program is a set of data for this database

**Facts**

- A fact is a term followed by a colon.
- This is a Prolog program of six facts
- parent is a predicate of arity 2
- Natural interpretation: facts about families: Kim is the parent of Holly, etc

<table>
<thead>
<tr>
<th>fact</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(kim, holly)</td>
<td>esther heritage jean</td>
</tr>
<tr>
<td>parent(margaret, kim)</td>
<td>margaret jean</td>
</tr>
<tr>
<td>parent(margaret, kent)</td>
<td>kim kent</td>
</tr>
<tr>
<td>parent(esther, margaret)</td>
<td>esther jean</td>
</tr>
<tr>
<td>parent(herbert, margaret)</td>
<td>herbert margaret</td>
</tr>
<tr>
<td>parent(herbert, jean)</td>
<td>herbert jean</td>
</tr>
</tbody>
</table>

**SWI-Prolog**

- Welcome to SWI-Prolog (Version 3.4.2)
- Copyright (c) 1990-2000 University of Amsterdam.
- Copy policy: GPL-2 (see www.gnu.org)
- For help, use ?- help(Topic), or ?- apropos(Word).
- ?-

**Simple Queries**

- Now we can ask the language system to prove something = make a query
- The answer will be Yes or No
- (Some queries, like consult, are executed only for their side-effects)
Queries With Variables

?- parent (P, jean).
\[ P = \text{herbert} \]
Yes
?- parent (P, esther).
No

The Prolog system shows the bindings necessary to prove the query.
The binding makes a query term unify with a fact term from the program.

Unification

n Two Prolog terms \( t_1 \) and \( t_2 \) unify if there is some substitution \( \sigma \) (their unifier) that makes them identical: \( \sigma(t_1) = \sigma(t_2) \)
\[ \sigma(a, b) \text{ and } \sigma(a, X) \text{ unify if } \sigma \text{ maps } a \text{ to } b \]
\[ \sigma(\text{parent}(P, \text{child})) \text{ and } \sigma(\text{parent}(\text{child}, \text{parent})) \text{ do not unify} \]
\[ \sigma(\text{parent}(X, Z)) \text{ and } \sigma(X, \text{parent}(X, Z)) \text{ do not unify} \]

Substitutions

n A substitution is a function that maps variables to terms:
\[ \sigma = \{X \mapsto a, P \mapsto \text{herbert}\} \]
This \( \sigma \) maps \( X \) to \( a \) and \( P \) to herbert.
The result of applying a substitution to a term is an instance of the term.

Flexibility

n More flexible than Java
n Normally, variables can appear in any or all positions in a query:
\[ \text{parent} (\text{Parent}, \text{jean}) \]
\[ \text{parent} (\text{esther}, \text{Child}) \]
\[ \text{parent} (\text{Parent}, \text{Child}) \]
\[ \text{parent} (\text{Person}, \text{Person}) \]

Conjunctions

n A conjunctive query has a list of query terms separated by commas
n The Prolog system tries prove them all (using a single set of bindings)
Multiple Solutions

There might be more than one way to prove the query

By typing ; rather than Enter, you ask the Prolog system to find more

Rules

A rule says how to prove something: to prove the head, prove the conditions

To prove greatgrandparent(GGP, GCC), find some GP and P for which you can prove parent(GGP, GP), then parent(GP, P) and then finally parent(P, GCC)

Example

This shows the initial query and final result

Internally, there are intermediate goals:

The first goal is the initial query

The next is what remains to be proved after transforming the first goal using one of the clauses (in this case, the greatgrandparent rule)

And so on, until nothing remains to be proved
Scope

Same relation, defined indirectly
Note that both clauses use a variable \( P \)
First occurrence of a variable serves as definition
The scope of the definition of a variable is the clause that contains it

Recursion

Recursion plays a central role in Prolog
For example take the relation
parent \((X, Y)\)
Such a relation can be generalized to \( X \) is ancestor of \( Y \), to operate over as many generations as necessary.

Recursive Rules

\( X \) is an ancestor of \( Y \) if:
- Base case: \( X \) is a parent of \( Y \)
- Recursive case: there is some \( Z \) such that \( Z \) is a parent of \( Y \), and \( X \) is an ancestor of \( Z \)
Prolog tries rules in the order you give them, so put base-case rules and facts first

Core Syntax Of Prolog

- You have seen the complete core syntax:
  - \(<\text{clause}> ::= <\text{fact}> | <\text{rule}>\)
  - \(<\text{fact}> ::= <\text{term}> : - <\text{termlist}>\)
  - \(<\text{termlist}> ::= <\text{term}> | <\text{term}>, <\text{termlist}>\)
- There is not much more syntax for Prolog than this: it is a very simple language

Let’s practice with a joke

Monty Python and the Holy Grail (Scene 5, The Witch Scene)

Script available from:
http://www.calvin.edu/~rpreim/courses/m156/F99/prolog/duck.txt
The Prolog program

```
witch(X) :- burns(X), female(X).
burns(X) :- wooden(X).
wooden(X) :- floats(X).
floats(X) :- same_weight(duck, X).
female(girl). /* by observation */
same_weight(duck,girl). /* by experiment */
? witch(girl).
```

Logic programming
- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

The Procedural Side
- greatgrandparent(GGP, GCC) :-
  parent(GGP, GP), parent(GP, P), parent(P, GCC).
- A rule says how to prove something:
  To prove greatgrandparent(GGP, GCC), find some
  G and P for which you can prove
  parent(GGP, GP), then parent(GP, P) and then
  finally parent(P, GCC).
- A Prolog program specifies proof procedures for queries

The Declarative Side
- A rule is a logical assertion:
  For all bindings of GGP, GP, P, and GCC, if
  parent(GGP, GP) and parent(GP, P) and
  parent(P, GCC), then
  greatgrandparent(GGP, GCC).
- Just a formula — it doesn’t say how to do anything — it just makes an
  assertion:

\[ \forall GGP, GP, P, GCC. \quad \text{parent}(GGP, GP) \land \text{parent}(GP, P) \land \text{parent}(P, GCC) \]
\[ \Rightarrow \text{greatgrandparent}(GGP, GCC) \]

Declarative Languages
- Each piece of the program corresponds to a simple mathematical abstraction
  - Prolog clauses – formulae in first-order logic
  - ML fun definitions – functions
- Many people use declarative as the opposite of imperative, including both logic
  languages and functional languages
Declarative Advantages
- Imperative languages are doomed to subtle side-effects and interdependencies
- Simpler declarative semantics makes it easier to develop and maintain correct programs
- Higher-level, more like automatic programming: describe the problem and have the computer write the program

Prolog Has Both Aspects
- Partly declarative
  - A Prolog program has logical content
- Partly procedural
  - A Prolog program has procedural concerns: clause ordering, condition ordering, side-effecting predicates, etc.
- It is important to be aware of both

Logic programming
- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

Backtracking
- The language system has to look for a solution to a query and guarantee termination – difficult task.
- Prolog uses a simple backtracking strategy
- If efficiency is important, backtracking can be controlled.

Backtracking
- Simple backtracking: depth-first tree search with subgoal evaluation from left to right
- User gives a goal. Prolog starts to satisfy the list of subgoals and if failure occurs it backtracks and tries an alternative set of goals

Backtracking
- Prolog explores all possible targets of each call, until it finds as many successes as the caller requires or runs out of possibilities
Backtracking Example

1. loves(john, jane).
2. loves(john, hilda).
3. loves(bill, jane).
4. loves(james, jane).
5. loves(mary, bill).
6. loves(jane, james).
7. goodmatch(X, Y) :- loves(X, Y), loves(Y, X).

?- goodmatch(A, B)

Trace

Backtracking Example

1. loves(john, jane).
2. loves(john, hilda).
3. loves(bill, jane).
4. loves(james, jane).
5. loves(mary, bill).
6. loves(jane, james).
7. goodmatch(X, Y) :- loves(X, Y), loves(Y, X).

?- goodmatch(A, B)

What is the disadvantage of Prolog automatic backtracking?

- Controlled backtracking - cut
  - The cut operation ! can stop backtracking.
  - Prevents from trying alternatives that cannot succeed.
  - Useful in mutually exclusive clauses

Same example, with cut

1. loves(john, jane).
2. loves(bill, jane).
3. loves(james, jane).
4. loves(mary, bill).
5. loves(jane, james).
6. goodmatch(X, Y) :- loves(X, Y), loves(Y, X), !.

Disadvantage of cuts

- Declarative and procedural meaning may differ
- Green and red cuts
Logic programming
- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

Second look at Prolog
- Operators
- Lists
- Anonymous
- Negation

Operators
- Prolog has some predefined operators (and the ability to define new ones)
- An operator is just a predicate for which a special abbreviated syntax is supported

The = Predicate
- The goal \( (X, Y) \) succeeds if and only if \( X \) and \( Y \) can be unified:

\[
\begin{align*}
\text{Yes} & \quad \text{if} \\
& \text{Since } = \text{ is an operator, it can be and usually is written like this:}
\end{align*}
\]

Arithmetic Operators
- Predicates +, -, *, and / are operators too, with the usual precedence and associativity

Terms are Not Evaluated
- The term is still \( + (1, * (2, 3)) \)
- It is not evaluated
Lists in Prolog

- A bit like ML lists
- The atom `[ ]` represents the empty list
- A predicate `.` corresponds to ML’s `::` operator

<table>
<thead>
<tr>
<th>ML expression</th>
<th>Prolog term</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ ]</code></td>
<td><code>.(</code></td>
</tr>
<tr>
<td><code>3 :: [ ]</code></td>
<td><code>.(</code> <code>. </code></td>
</tr>
<tr>
<td><code>None</code></td>
<td><code>.(</code> <code>. </code></td>
</tr>
<tr>
<td>No equivalent.</td>
<td><code>.(</code> <code>. </code></td>
</tr>
</tbody>
</table>

List Notation

- ML-style notation for lists
- These are just abbreviations for the underlying term using the `.` predicate
- Prolog usually displays lists in this notation

<table>
<thead>
<tr>
<th>List notation</th>
<th>Term denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ ]</code></td>
<td><code>[ ]</code></td>
</tr>
<tr>
<td><code>[1]</code></td>
<td><code>(1, </code></td>
</tr>
<tr>
<td><code>[1, 2, 3]</code></td>
<td><code>(1, </code>(2, <code>(3, [ ])))</code></td>
</tr>
<tr>
<td><code>[1, parent X Y]</code></td>
<td><code>(1, </code>(parent X Y, <code>[ ])))</code></td>
</tr>
</tbody>
</table>

Example

```
?- X = .(1, .(2, .(3, [ ]))).
X = [1, 2, 3]
Yes
?- (X, Y) = [1, 2, 3].
X = 1
Y = [2, 3]
Yes
```

List Notation With Tail

- Last in a list can be the symbol `|` followed by a final term for the tail of the list
- Useful in patterns: `[1, 2 | X]` unifies with any list that starts with `1, 2` and binds `X` to the tail

```
?- [1, 2 | X] = [1, 2, 3, 4, 5].
X = [3, 4, 5]
Yes
```

The append Predicate

```
?- append([1, 2], [3, 4], Z).
Z = [1, 2, 3, 4]
Yes
```

- Predefined `append(X, Y, Z)` succeeds if and only if `Z` is the result of appending the list `Y` onto the end of the list `X`

append is Not Just A Function

```
?- append(X, [3, 4], [1, 2, 3, 4]).
X = [1, 2]
Yes
```

- `append` can be used with any pattern of instantiation (that is, with variables in any positions)
Not Just A Function

```prolog
?- append(X, Y, [1, 2, 3]).
X = [ ]
Y = [1, 2, 3] ;
X = [1]
Y = [2, 3] ;
X = [1, 2]
Y = [3] ;
X = [1, 2, 3]
Y = [1] ;
No
```

Other Predefined List Predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>member(X, Y)</td>
<td>Provable if the list Y contains the element X.</td>
</tr>
<tr>
<td>select(X, Y, Z)</td>
<td>Provable if the list Y contains the element X, and Z is the same as Y but with one instance of X removed.</td>
</tr>
<tr>
<td>nth0(X, Y, Z)</td>
<td>Provable if X is an integer, Y is a list, and Z is the Xth element of Y, counting from 0.</td>
</tr>
<tr>
<td>length(X, Y)</td>
<td>Provable if X is a list of length Y.</td>
</tr>
</tbody>
</table>

- All flexible, like `append`
- Queries can contain variables anywhere

Using `select`

```prolog
?- select(2, [1, 2, 3], Z).
Z = [1, 3] ;
No
?- select(2, Y, [1, 3]) .
Y = [2, 1, 3] ;
Y = [1, 2, 3] ;
Y = [1, 3, 2] ;
No
```

The `reverse` Predicate

```prolog
?- reverse([1, 2, 3, 4], Y).
Y = [4, 3, 2, 1] ;
No
```

- Predefined `reverse(X, Y)` unifies Y with the reverse of the list X
- Not flexible

The Anonymous Variable

- The variable `_;` is an anonymous variable
- Every occurrence is bound independently of every other occurrence
- In effect, much like ML's `_;` it matches any term without introducing bindings

Example

```prolog
tailof(_, A, A).
```

- This `tailof(X, Y)` succeeds when X is a non-empty list and Y is the tail of that list
- Don’t use this, even though it works:
  ```prolog
tailof(_, Head, A, A).
  ```
The **not** Predicate

\[ \text{not}(\text{number}(1,1,2,3)). \]
\[ \text{yes} \]
\[ \text{not}(\text{number}(4,1,2,3)). \]
\[ \text{yes} \]

- For simple applications, it often works quite a bit logical negation
- But it has an important procedural side...

---

**Negation**

- Prolog uses the closed-world assumptions; it assumes that all facts about the world are included in the model.
- Ex: To prove `not married (fred)`, Prolog attempts to prove `married (fred)`.
- `not married (fred)` succeeds if the system fails to prove that `fred` is married.
- It can happen in 2 cases:
  - Bill is really not married
    - System cannot prove that bill is married
- Conclusion: take care in interpreting the answer to a negation question!!

---

**Example**

```
\text{Sibling}(X, Y) :=
\text{not}(X=Y),
\text{parent}(P, X),
\text{parent}(P, Y),
\text{not}(X=Y).
```

\[ \text{Sibling}(\text{kim}, \text{kent}). \]
\[ \text{yes} \]
\[ \text{Sibling}(\text{kim}, \text{kim}). \]
\[ \text{no} \]
\[ \text{Sibling}(\text{kent}, \text{kim}). \]
\[ \text{no} \]

---

**Logic programming**

- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

---

**The problem: A Classic Riddle**

- A man travels with wolf, goat and cabbage
- Wants to cross a river from west to east
- A rowboat is available, but only large enough for the man plus one possession
- Wolf eats goat if left alone together
- Goat eats cabbage if left alone together
- How can the man cross without loss?
Configurations

- Represent a configuration of this system as a list showing which bank each thing is on in this order: man, wolf, goat, cabbage
- Initial configuration: \([w, w, w, w]\)
- If man crosses with wolf, new state is \([e, e, w, w]\) — but then goat eats cabbage, so we can’t go through that state
- Desired final state: \([e, e, e, e]\)

Moves Transform Configurations

- Each move transforms one configuration to another
- In Prolog, we will write this as a predicate:
  \[
  \text{move(Config, Move, NextConfig)}
  \]
  where:
  - Config is a configuration (like \([w, w, w, w]\))
  - Move is a move (like wolf)
  - NextConfig is the resulting configuration (in this case, \([e, e, w, w]\))

The move Predicate

\[
\begin{align*}
\text{move} & \left( [X, X, \text{Goat}, \text{Cabbage}], \text{wolf}, [Y, Y, \text{Goat}, \text{Cabbage}] \right) & : \text{change}(X, Y) \\
\text{move} & \left( [X, \text{Wolf}, X, \text{Cabbage}], \text{goat}, [Y, \text{Wolf}, Y, \text{Cabbage}] \right) & : \text{change}(X, Y) \\
\text{move} & \left( [X, \text{Wolf}, \text{Goat}, X], \text{cabbage}, [Y, \text{Wolf}, \text{Goat}, Y] \right) & : \text{change}(X, Y) \\
\text{move} & \left( [X, \text{Wolf}, \text{Goat}, C], \text{nothing}, [Y, \text{Wolf}, \text{Goat}, C] \right) & : \text{change}(X, Y) \\
\end{align*}
\]
Safe Configurations

A configuration is safe if

- At least one of the goat or the wolf is on the same side as the man, and
- At least one of the goat or the cabbage is on the same side as the man

\[
\text{oneEq}(X, X, \_). \\
\text{oneEq}(X, \_, X).
\]

\[
\text{safe([Man, Wolf, Goat, Cabbage]) : :-} \\
\text{oneEq(Man, Goat, Wolf),} \\
\text{oneEq(Man, Goat, Cabbage).}
\]

Solutions

A solution is a starting configuration and a list of moves that takes you to \([e, e, e, e]\), where all the intermediate configurations are safe

\[
\text{solution([e,e,e,e], []).} \\
\text{solution(Configuration, [Move|Rest]) : :-} \\
\text{move(Configuration, Move, NextConfiguration),} \\
\text{safe(NextConfiguration),} \\
\text{solution(NextConfiguration, Rest).}
\]

Prolog Finds A Solution

\[
\text{length(X, 7), solution([w,w,w,w], X).} \\
X = \{\text{goat, nothing, wolf, goat, cabbage, nothing, goat}\}
\]

Note: without the \text{length}(X, 7) restriction, Prolog would not find a solution

It gets lost looking at possible solutions like \[\text{[goat, goat, goat, goat, goat…]}\]

What Prolog Is Good For

- The program specified a problem logically
- It did not say how to search for a solution to the problem – Prolog took it from there
- That’s one kind of problem Prolog is especially good for

Conclusion

- Logic programming is almost exclusively carried out by Prolog
- Prolog is a declarative language and is goal oriented
- A Prolog program consists of facts and rules
- The Prolog language system solves a query by matching. If the goal fails an alternative is tried by backtracking
- Efficiency can be improved with cuts
- Prolog is suited to solve logic problems in AI