CMSC 330: Organization of Programming Languages

Logic Programming with Prolog
Background

• 1972, University of Aix-Marseille

• Original goal: Natural language processing

• At first, just an interpreter written in Algol
  – Compiler created at Univ. of Edinburgh
More Information on Prolog

- Various tutorials available online
- Links on webpage
  - Eclipse plug-in ProDT available on-line
Logic programming

- At a high level, logic programs model the relationship between ‘objects’
  - Programmer specifies these relationships at a high level
  - The language builds a database
  - The programmer then queries this database
  - The language searches for answers
Features of Prolog

• Declarative
  – Specify what goals you want to prove, not how to prove them (mostly)
• Rule based
• Dynamically typed
• Several built-in datatypes
  – Lists, numbers, records, … but no functions
• Several other logic programming languages
  – Datalog is simpler; CLP and λProlog more feature-ful
  – Erlang borrows some features from Prolog
/* A small Prolog program */

female(alice).
male(bob).
male(charlie).
father(bob, charlie).
mother(alice, charlie).

% “X is a son of Y”
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).

Use /* */ for comments, or % for 1-liners
Period ends statements

Lowercase logically terminates
Program consists of facts and rules
Uppercase denotes variables
Running Prolog (interactive mode)

Expressions can also be typed and evaluated at the top-level:

?- ['father_mother.pl'].
% father_mother.pl compiled 0.00 sec, 8 clauses
true.

?- son(X,Y).
X = charlie,
Y = bob;
X = charlie,
Y = alice.

?- make.
father_mother compiled 0.00 sec, 3 clauses
true.
Outline

• Syntax, terms, and more examples
• Lists
• Arithmetic / evaluation / precedence (= vs. is)
• How Prolog resolves queries, informally
• Formal definition of query evaluation
  – Notions of ordering, backtracking, unification
• Cuts, negation
Prolog syntax and terminology

• Terms
  – **Atoms**: begin with a lowercase letter
    
    | horse      | underscores_ok | numbers2 |
  
  – **Numbers**
    
    | 123      | -234    | -12e-4   |
  
  – **Variables**: begin with uppercase or _
    
    | X        | Biggest_Animal | _the_biggest1 |
    |          |             |               |
    |          |             |               |
    |          |             |               |
    |          |             |               |
  
  – **Compound terms**: functor(arguments)
    
    | bigger(horse, duck) |
    | bigger(X, duck)     |
    | f(a, g(X, _), Y, _) |

No blank spaces between functor and (arguments)
Prolog syntax and terms (cont’d)

• Clauses
  – Facts: define predicates, terminated by a period
    bigger(horse, duck).
    bigger(duck, gnat).
    Intuitively: “this particular relationship is true”
  – Rules: Head :- Body
    is_bigger(X,Y) :- bigger(X,Y).
    is_bigger(X,Y) :- bigger(X,Z), is_bigger(Z,Y).
    Intuitively: “Head if Body”, or “Head is true if each of
    the subgoals can be shown to be true”

• A program is a sequence of clauses
Prolog syntax and terms (cont’d)

• Queries
  – To “run a program” is to submit queries to the interpreter
  – Same structure as the body of a rule:
    • Predicates separated by commas, ended with a period
  – Prolog tries to determine whether or not the predicates are true

?- is_bigger(horse, duck).
?- is_bigger(horse, X).

“Does there exist a substitution for X such that is_bigger(horse,X)?”
Unification – the sine qua non of Prolog

• Two terms unify *if and only if*
  
  – They are identical

  
  ```
  ?- gnat = gnat.
  true.
  ```

  – They can be made identical by *substituting* variables

  ```
  ?- is_bigger(X, gnat) = is_bigger(elephant, gnat).
  X = elephant.  This is the *substitution*: what X must
  be for the two terms to be identical.
  ```

  ```
  ?- pred(X, 2, 2) = pred(1, Y, X)
  false
  ```

  Sometimes there are multiple possible substitutions; Prolog can be asked to enumerate them all

  ```
  ?- pred(X, 2, 2) = pred(1, Y, _)
  X = 1,
  Y = 2.
  ```
Prolog terminology

• A query, goal, or term where variables do not occur is called ground; else it’s nonground
  – foo(a,b) is ground; bar(X) is nonground

• A substitution $\theta$ is a partial map from variables to terms where $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$
  – Variables are terms, so a substitution can map variables to other variables, but not to themselves

• A is an instance of B if there is a substitution such that $A = B\theta$

• C is a common instance of A and B if it is an instance of A and an instance of B
Goal execution

- When submitting a query, we ask Prolog to substitute variables as necessary to make it true
- Prolog performs goal execution to find a solution
  - Start with the goal
  - Try to unify the head of a rule with the current goal
  - The rule hypotheses become subgoals
    - Substitutions from one subgoal constrain solutions to the next
  - If it reaches a deadend, it backtracks
    - Tries a different rule
  - When it can backtrack no further, it reports false
Goal execution (cont’d)

• Consider the following:
  – “All men are mortal”
    \[
    \text{mortal}(X) :- \text{man}(X).
    \]
  – “Socrates is a man”
    \[
    \text{man}(\text{socrates}).
    \]
  – “Is Socrates mortal?”
    \[
    ?- \text{mortal}(\text{socrates}).
    \]
    \[
    \text{true}.
    \]

• How did Prolog infer this?

  1. Sets \text{mortal}(\text{socrates}) as the initial goal
  2. Sees if it unifies with the head of any clause:
    \[
    \text{mortal}(\text{socrates}) = \text{mortal}(X).
    \]
  3. \text{man}(\text{socrates}) becomes the new goal (since \(X=\text{socrates}\))
  4. Recursively scans through all clauses, backtracking if needed …
Tracing

- `trace` lets you step through a goal’s execution
  - `notrace` turns it off

```prolog
my_last(X, [X]).
my_last(X, [__|T]) :-
    my_last(X, T).
?- trace.
true.
[trace]  ?- my_last(X, [1,2,3]).
  Call: (6) my_last(_G2148, [1, 2, 3]) ? creep
  Call: (7) my_last(_G2148, [2, 3]) ? creep
  Call: (8) my_last(_G2148, [3]) ? creep
  Exit: (8) my_last(3, [3]) ? creep
  Exit: (7) my_last(3, [2, 3]) ? creep
  Exit: (6) my_last(3, [1, 2, 3]) ? creep
X = 3
```
Style

One predicate per line

blond(X) :-
  father(Father, X),
  blond(Father), % father is blond
  mother(Mother, X),
  blond(Mother). % and mother is blond

Descriptive variable names

Inline comments with % can be useful
More syntax: Built-in predicates

- Equality (a.k.a. *unification*)
  \[ X = Y \quad f(1,X,2) = f(Y,3,\_1) \]

- **fail** and **true**

- “Consulting” (loading) programs
  \[ ?- \text{consult('file.pl')} \quad ?- ['file.pl'] \]

- Output/Input
  \[ ?- \text{write('Hello world'), nl} \quad ?- \text{read(X)} \]

- (Dynamic) type checking
  \[ ?- \text{atom(elephant)} \quad ?- \text{atom(Elephant)} \]

- **help**
Lists in Prolog

- \([a, b, 1, ‘hi’, [X, 2]]\)
- But really represented as compound terms
  - \([]\) is an atom
  - \([a, b, c]\) is represented as \(.(a, .(b, .(c, [])))\)
- Matching over lists
  ?- \([X, 1, Z] = [a, _, 17]\)
  \(X = a,\)
  \(Z = 17.\)
List deconstruction

• Syntactically similar to Ocaml: \([H|T]\) like \h::t
  
  \(- [\text{Head} \mid \text{Tail}] = [a,b,c].
  \head = a,
  \tail = [b, c].

  \(- [1,2,3,4] = [\_, X \mid \_].
  \X = 2

• This is sufficient for defining complex predicates

• Let’s define \text{concat}(L1, L2, C):

  \(- \text{concat}([a,b,c], [d,e,f], X).
  \X = [a,b,c,d,e,f].
Example: concatenating lists

• To program this, we define the “rules” of concatenation
  – If L1 is empty, then C = L2
    \[
    \text{concat( } [], \text{ L2, L2 ).}
    \]
  – Prepending a new element to L1 prepends it to C, so long as C is the concatenation of L1 with some L2
    \[
    \text{concat( } [E \mid L1], \text{ L2, [E \mid C] ) :-}
    \]
    \[
    \text{concat(L1, L2, C).}
    \]

• … and we’re done
Why is the return value an argument?

• Now we can ask *what inputs lead to an output*

```prolog
?- concat(X, Y, [a,b,c]).

\[
\begin{cases}
X = [], \\
Y = [a, b, c] ; \\
X = [a], \\
Y = [b, c] ; \\
X = [a, b], \\
Y = [c] ; \\
X = [a, b, c], \\
Y = [] ;
\end{cases}
\]
```
Built-in list predicates

• length
  ?- length([a, b, [1,2,3]], Length).
  Length = 3.

• member
  ?- member(duey, [huey, duey, luey]).
  true.

• select
  ?- select(duey, [huey, duey, luey], X).
  X = [huey, luey].

• See documentation for more
  – http://www.swi-prolog.org/pldoc/refman/ Appendix A
Matching and evaluation

- \(?- 9 = 9.\)
  - true.
- \(?- 7 + 2 = 9.\)
  - false.

- Why? Because = is for \textit{matching} and these atoms do not match (7+2 is a compound term)
- Prolog does not evaluate either side of = before trying to match
- To tell Prolog to first evaluate, we use \texttt{is}
The `is` operator

- For arithmetic operations
- “LHS is RHS”
  - First *evaluate* the RHS (and RHS only!) to value V
  - Then match: LHS = V
- Examples

  - `- 9 is 7+2.`
    true.
  - `- 7+2 is 9.`
    false.
  - `- X = 7+2.`
    `X = 7+2.`
  - `- X is 7+2.`
    `X = 9.`
Logic program evaluation, formally

Input: A goal $G$ and a program $P$
Output: $G\theta$: an instance of $G$ deduced from $P$, or fail

Algorithm 1:

– Initialize the resolvent to $G$
– While the resolvent $A_1,\ldots,A_n$ is not empty:
  • Choose (1) a goal $A_i$ from resolvent, and
    (2) a clause $A : - B_1,B_2,\ldots,B_k$ in $P$, and
    (3) a substitution $\theta$ such that $A\theta=A_i\theta$
    – If no such clause exists, exit the while loop
  • Set $G$ to $G\theta$ and resolvent to $(A_1,\ldots,A_{i-1},B_1,\ldots,B_k,A_{i+1},\ldots,A_n)\theta$
    – If resolvent is empty, output $G$, else fail
Two notes about Algorithm 1

• No backtracking
  – The algorithm chooses some clause (step (2)) from $P$ but this clause may end up leading to failure where another clause might have succeeded

• No ordering
  – The algorithm can pick any goal from the resolvent – they are not resolved in any particular order

• In contrast: Prolog chooses clauses in order, and employs backtracking to try other clauses on failure. We’ll see this later
**Prolog’s algorithm** `Solve()`

\[
\text{Solve}(\text{goal } G, \text{ program } P, \text{ substitution } \theta) =
\]

- Suppose \( G \) is \( A_1, \ldots, A_n \). Choose goal \( A_1 \).
- For each clause \( A \leftarrow B_1, B_2, \ldots, B_k \) in \( P \),
  - if \( \theta_1 \) is the \textit{mgu} of \( A \) and \( A_1 \theta \) then
    - If \( \text{Solve}(\{B_1, \ldots, B_k, A_2, \ldots, A_n\}, P, \theta \cdot \theta_1) = \text{some } \theta' \) then \textbf{return } \theta'
    - (else it has failed, so we continue the for loop)
  - (else unification has failed, so try another rule)
- If loop exits return \textit{fail}
- Output: \( \theta \) \text{ s.t. } G\theta \text{ can be deduced from } P, \text{ or } \textit{fail}
Most general unifier (in pseudo-Ocaml)

- \text{mgu}(t_1, t_2) =
  - match \( t_1, t_2 \) with
    - \( c, c \mid X, X \implies \cdot \)
    - \( X, \_ \) when \( \neg \text{occurs}(X, t_2) \implies [X \mapsto t_2] \)
    - \( \_, Y \) when \( \neg \text{occurs}(Y, t_1) \implies [Y \mapsto t_1] \)
    - \( f(t_1^1, t_2^2, \ldots, t_n^n), f(t_1^1, t_2^2, \ldots, t_n^n) \implies \) (for some \( n>0 \))
      - let \( \theta_1 = \text{mgu}(t_1^1, t_2^1) \) in \( \) (else fail if no \( \theta_1 \) etc)
      - let \( \theta_2 = \text{mgu}(t_2^2 \theta_1, t_2^2 \theta_1) \) in \( \ldots \)
      - let \( \theta_n = \text{mgu}(t_1^{n-1} \theta_1 \cdot \ldots \cdot t_n^{n-1}, t_2^{n-1} \theta_1 \cdot \ldots \cdot \theta_{n-1}) \) in \( \theta_1 \cdot \ldots \cdot \theta_{n-1} \cdot \theta_n \)
    - \( \_ \implies \text{fail} \)
Occurs check

• \textit{occurs}(X,t) if and only if
  – t is X, or
  – t is some term \( f(t_1,\ldots,t_n) \) and there exists some \( i \) such that \( 1 \leq i \leq n \) and \textit{occurs}(X,t_i).

• This is used during unification to avoid cyclic substitutions
! : a.k.a. “cut”

- When a ! is reached, it succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in
  - Suppose we have clause C which is
    \[ A \leftarrow B_1, \ldots, B_k, !, \ldots, B_n. \]
  - If the current goal unifies with \( A \), and \( B_1, \ldots, B_k \) further succeed, the program is committed to the choice of \( C \) for the goal.
    - If any \( B_i \) for \( i > k \) fail, backtracking only goes as far as the cut.
    - If the cut is reached when backtracking, the goal fails
Prolog’s algorithm Solve() with cut

Solve(goal G, program P, substitution θ) =

• Suppose G is A₁,..,Aₙ. Choose goal A₁.
• For each clause A :- B₁,B₂,...,Bₖ in P,
  – if θ₁ is the mgu of A and A₁θ then
    • If Solve({B₁,...,Bₖ,A₂,...,Aₙ}, P, θ·θ₁) = some θ’ then return θ’
    • (else it has failed, so we continue the for loop)
  – (else unification has failed, so try another rule)

• If loop exits return fail
  – If A₁ was a cut !, then the entire (recursive) process abandoned

• Output: θ s.t. Gθ can be deduced from P, or fail
Negation as failure

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

- If $X$ succeeds, then the cut is reached, committing it; `fail` causes the whole thing to fail.
- If $X$ fails, then the second rule is reached, and the overall goal succeeds.
  - FYI, $X$ here is a \textit{meta-variable}, referring to an arbitrary goal.
  - Depends crucially on rule order.
What exactly is cut doing?

- Prunes all clauses below it
- Prunes alternative solutions to its left
- Does not affect the goals to its right

merge([X|Xs], [Y|Ys], [X|Zs]) :-
  X < Y, !,
  merge(Xs, [Y|Ys], Zs).

merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
  X =:= Y, !,
  merge(Xs,Ys,Zs).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
  X > Y, !,
  merge([X|Xs],Ys,Zs).

merge(Xs, [], Xs) : !
merge([], Ys, Ys) : !.
Why use cuts?

• Save time and space, or eliminate redundancy
  – These prune useless branches in the search tree
  – These are green cuts

• Guide to the search to a different solution
  – These change the meaning of the program
  – These are red cuts
Consequences of negation as failure

• The proof search for the failing assertion $G$ must terminate
  – Or not $G$ will not terminate

• Rule order may matter
  – `unmarried_student(X) :- not(married(X)), student(X).`
  – `student(bill).`
  – `married(joe).`
  – `? unmarried_student(X) fails`
    • However, it succeeds if you change the first clause to
      `unmarried_student(X) :- student(X), not married(X)`