Prolog

Principles of Programming Languages

CSE 526

Introduction

Logic and Programs

- “All men are mortal; Socrates is a man; Hence Socrates is mortal”

\[ \forall X. \text{man}(X) \Rightarrow \text{mortal}(X) \]

\[ \text{man(socrates)} \]

- Predicate logic
  - Predicates (e.g. \text{man, mortal}) which define sets.
  - Atoms (e.g. \text{socrates}) which are data values
  - Variables (e.g. \text{X}) which range over data values
  - Rules (e.g. \forall X. \text{man}(X) \Rightarrow \text{mortal}(X)) which define relationships
  between predicates.
Inference Rules and Logic Programs

Logic “Program”:
\[
\begin{align*}
\text{man}(\text{socrates}) & \quad \text{R-MAN} \\
\text{man}(x) & \quad \frac{\text{man}(x)}{\text{mortal}(x)} & \text{R-MORTAL}
\end{align*}
\]

man(socrates).
mortal(X) :- man(X).

Queries:
?- mortal(socrates).
  yes
?- mortal(X).
  \(X=socrates\);
  no

Programming in Logic

- Early development: Kowalski & van Emden (Edinburgh); Colmerauer (Marseilles) (early ’70s)
- First efficient implementation: WAM of David H.D. Warren (Edinburgh) (mid ’70s).
- Later developments:
  - Constraint Logic Programming: for applications in AI, planning, scheduling, etc. Jaffar & Lassez (IBM Watson)
  - Memoization: Tamaki & Sato (Tokyo); Warren et al (Stony Brook)
Prolog Systems

- SWI Prolog (www.swi-prolog.org)
  - Can be obtained for free and installed on Windows, Linux, Mac.
  - Has a good development environment (command completion, help, graphical debugger, etc.)
  - On compute* (Unix) servers: ~cram/bin/swipl

- XSB Prolog (xsb.sourceforge.net)
  - Can be obtained for free and installed on Windows, Linux, Mac.
  - Supports a powerful extension (memoization) to Prolog
  - Command-line interface (e.g. no graphical debugger)
  - On compute* (Unix) servers: ~cram/bin/xsb

Using Prolog Systems

- Prolog programs are in files with “.pl” extension (“.P” for XSB)
- Prolog systems typically support an interactive mode.
- “[filename].” to compile and load a program in filename.pl (filename.P in XSB).
- “halt.” to exit the system.
Logic Programs

- Programs are a set of rules (also called clauses).
- Predicates in a logic program are analogous to procedures in imperative programs.
- One or more rules are used to define a predicate.
- Example:
  
  \[
  \text{inc}(X,Y) :- Y \text{ is } X+1.
  \]

  - \(X\) and \(Y\) are variables.
  - inc is a predicate.
  - The predicate is defined using a single rule.

\[
\text{inc}(X,Y) :- Y \text{ is } X+1.
\]

  - “:-” separates the body of the rule from its head.
  - “\(X\)” and “\(Y\)” are also “parameters” of the predicate.
    - In this case, \(X\) is the input parameter, and \(Y\) is the return parameter (where the return values are stored).
  - “\(Y \text{ is } X+1\)” defines \(Y\) in terms of \(X\).
  - The period (“.”) marks the end of a rule.
  - The predicate is called by giving values to its parameters. e.g.
    - \(\text{inc}(6, B)\) returns with \(B=7\).
    - \(\text{inc}(11, B)\) returns with \(B=12\).
Syntax of Prolog

- **Variables** are identifiers that begin with an upper case letter or underscore.
  - An underscore, by itself, represents an *anonymous variable*.
- **Predicate** names (and later, data structure symbols) are identifiers that begin with a lower case letter.
- All variables are *local* to the clause in which they occur.
- Different occurrences of the same variable in a clause denote the same data.
- Variables need not be declared, and have no type.

How Prolog Works (An Example)

```prolog
big(bear).
big(elephant).

brown(bear).

black(cat).

small(cat).

gray(elephant).

dark(Z) :- black(Z).
dark(Z) :- brown(Z).

dangerous(X) :- dark(X), big(X).
```
How Prolog Works (the procedure)

- A **query** is, in general, a conjunction of **goals**
- To prove $G_1, G_2, \ldots, G_n$:
  - Find a clause $H : -B_1, B_2, \ldots, B_k$ such that $G_1$ and $H$ match.
  - Under that substitution for variables, prove $B_1, B_2, \ldots, B_k, G_2, \ldots, G_n$.
  - If nothing is left to prove then the proof is complete. If there are no more clauses to match, the proof attempt fails.
How Prolog Works (an example)

To prove dangerous(Q):

1. Select dangerous(X) :- dark(X), big(X) and prove dark(Q), big(Q).
2. To prove dark(Q) select the first clause of dark, i.e. dark(Z) :- black(Z), and prove black(Q), big(Q).
3. Now select the fact black(cat) and prove big(cat).
   **This proof attempt fails!**
4. Go back to step 2, and select the second clause of dark, i.e. dark(Z) :- brown(Z), and prove brown(Q), big(Q).
5. Now select brown(bear) and prove big(bear).
6. Select the fact big(bear).
   **There is nothing left to prove, so the proof is complete**

Data Representation in Prolog

- Prolog has no notion of data types
- All data is represented as terms, which can be:
  - Variables
  - Non-variable Terms
    - Atomic data (Integers, floating point numbers, constants, ...)
    - Compound Terms (Structures)
Atomic Data

- **Numeric constants**: Integers, floating point numbers (e.g. 1024, -42, 3.1415, 6.023e23 ...)
- **Atoms**:
  - Strings of characters enclosed in single quotes (e.g. 'cram', 'Stony Brook')
  - Identifiers: sequence of letters, digits, underscore, beginning with a letter (e.g. cram, r2d2, x_24).

Data Structures

- If $f$ is an identifier and $t_1, t_2, \ldots t_n$ are terms, then $f(t_1, t_2, \ldots t_n)$ is a term.

- In the above, $f$ is called a *function symbol* (or *functor*) and $t_i$ is an *argument*.
- Structures are used to group related data items together (in some ways similar to *struct* in C and objects in Java).
- Structures are used to construct trees (and, as a special case, lists).
Trees

- Example: expression trees:
  \[ \text{plus}(\text{minus}(\text{num}(3), \text{num}(1)), \text{star}(\text{num}(4), \text{num}(2))) \]

- Data structures may have variables. And the same variable may occur multiple times in a data structure.

Matching

(We'll extend this to unification later)

- \( t_1 = t_2 \): find substitutions for variables in \( t_1 \) and \( t_2 \) that make the two terms identical.

Yes, with \( X = 1, Y = 4 \).
Matching (contd.)

Yes, with $X = 1$, $Y = 4$.

No! $X$ cannot be 1 and 4 at the same time.
Accessing arguments of a structure

- Matching is the predominant means for accessing a structures arguments.
- Let date(’Sep’, 1, 2005) be a structure used to represent dates, with the month, day and year as the three arguments (in that order).
- Then date(M, D, Y) = date(’Sep’, 1, 2005) makes $M = ’Sep’, D = 1, Y = 2005$.
- If we want to get only the day, we can write date(_, D, _) = date(’Sep’, 1, 2005). Then we get $D = 1$.

Lists

Prolog uses a special syntax to represent and manipulate lists.

- [1,2,3,4]: represents a list with 1, 2, 3 and 4, respectively.
- This can also be written as [1 | [2,3,4]]: a list with 1 as the head (its first element) and [2,3,4] as its tail (the list of remaining elements).
- If $X = 1$ and $Y = [2,3,4]$ then $[X|Y]$ is same as $[1,2,3,4]$.
- The empty list is represented by [ ].
- The symbol “|” (called cons) and is used to separate the beginning elements of a list from its tail.
  For example: $[1,2,3,4] = [1 | [2,3,4]]$
  $= [1 | [2 | [3,4]]]$ 
  $= [1,2 | [3,4]]$
Lists (contd.)

- Lists are special cases of trees.
  For instance, the list \([1,2,3,4]\) is represented by the following structure:

```
  4
 / \
3   [ ]
 /   \
2
 /   \
1
```

- The function symbol `.\(2\)` is the list constructor.
  \([1,2,3,4]\) is same as `(1, (2, (3, (4, []))))`.

---

Programming with Lists — I

First example: `member/2`, to find if a given element occurs in a list:

**The program:**

```
member(X, [X|Ys]).
member(X, [_|Ys]) :- member(X, Ys).
```

**Example queries:**

```
member(s, [l,i,s,t])
member(X, [l,i,s,t])
member(f(X), [f(1), g(2), f(3), h(4), f(5)])
```
Programming with Lists — II

append/3: concatenate two lists to form the third list.

**The program:**

```prolog
append([], L, L).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

**Example queries:**

```prolog
append([f,i,r], [s,t], L)
append(X, Y, [s,e,c,o,n,d])
append(X, [t,h], [f,o,u,r,t,h])
```

---

Programming with Lists — III

Define a predicate, `len/2` that finds the length of a list (first argument).

**The program:**

```prolog
len([], 0).
len([_|Xs], N+1) :- len(Xs, N).
```

**Example queries:**

```prolog
len([], X)
len([l,i,s,t], 4)
len([l,i,s,t], X)
```
Arithmetic

?- 1+2 = 3.

no

- In *Predicate logic*, the basis for Prolog, the only symbols that have a meaning are the predicates themselves.
- In particular, function symbols are uninterpreted: have no special meaning and can only be used to construct data structures.
- Meaning for arithmetic expressions is given by the *built-in* predicate “is”:
  - X is 1 + 2 succeeds, binding X to 3.
  - 3 is 1 + 2 succeeds.
  - General form: R is E where E is an expression to be evaluated and R is matched with the expression’s value.
  - Y is X + 1 will give an error if X does not (yet) have a value.

The list length example revisited

Define a predicate, `length/2` that finds the length of a list (first argument).

**The program:**

```prolog
length([], 0).
length([_|Xs], M) :- length(Xs, N), M is N+1.
```

**Example queries:**

```
length([], X)
length([l,i,s,t], 4)
length([l,i,s,t], X)
length(List, 4)
```
Conditional Evaluation

Consider the computation of \( n! \), i.e. the factorial of \( n \).

\[
\text{factorial}(N, F) :- \ldots
\]

- \( N \) is the input parameter; and \( F \) is the output parameter.
- The body of the rule specifies how the output is related to the input.
- For factorial, there are two cases: \( N \leq 0 \) and \( N > 0 \).
  - \( N \leq 0 \): \( F = 1 \)
  - \( N > 0 \): \( F = N \times (N - 1)! \)

\[
\text{factorial}(N, F) :-
  (N > 0
      -> N1 is N-1, factorial(N1, F1), F is N\times F1
      ; F = 1
  ).
\]

More Prolog Syntax

- Assignments with arithmetic expressions is done using the keyword “is”.
- If-then-else is written as ( cond -> then-part ; else-part )
- If more than one action needs to be performed in a rule, they are written one after another, separated by a comma.
- Arithmetic expressions are not directly used as arguments when calling a predicate; they are first evaluated, and then passed to the called predicate.
Arithmetic Operators

- Integer/Float operators: +, -, *, /
- Integer operators: mod, // (div)
- Int ↔ Float operators: floor, ceiling
- Comparison operators: <, >, =<, >=, =:=, =\= 

Sequences, revisited

append/3: concatenate two lists to form the third list (sometimes called conc/3).

The program:

```prolog
append([], L, L).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Example queries:

- `append([f,i,r], [s,t], L)`
- `append(X, Y, [s,e,c,o,n,d])`
- `append(X, [t,h], [f,o,u,r,t,h])`
Mystery Program

\[ m(X, X). \]
\[ m(X_1, X_5) :- a(X_1, X_2), m(X_2, X_3), b(X_3, X_4), m(X_4, X_5). \]
\[ a([0|Y], Y). \]
\[ b([1|Y], Y). \]

?- m([0,1,0,1,1], L).
   \[ L=[0,1,0,1,1] \]
   \[ L=[0,0,1,1] \]
   \[ L=[] \]

?- m([0,0,1,1,0], L).
   \[ L=[0,1,0,0,1,1] \]
   \[ L=[1,0] \]

Definite Clause Grammars

\[ m \rightarrow []. \]
\[ m \rightarrow a, m, b, m. \]
\[ a \rightarrow [0]. \]
\[ b \rightarrow [1]. \]

?- m([0,1,0,0,1,1], L).
   \[ L=[0,1,0,0,1,1], ... \]

?- phrase(m, [0,1,0,0,1,1]) \equiv m([0,1,0,0,1,1], [])
   \[ yes \]

?- phrase(m, L).
   \[ L=[] \]
   \[ L=[0,1] \]
   \[ L=[0,1,0,1] \]

...
Definite Clause Grammars (Magic?)

\[
\begin{align*}
  r([]) & \rightarrow [] . \\
  r([X|Xs]) & \rightarrow r(Xs), [X]. \\
\end{align*}
\]

Translated to:

\[
\begin{align*}
  r([], X, X). \\
  r([X|Xs], Z1, Z3) & : - r(Xs, Z1, Z2), Z2 = [X|Z3]. \\
\end{align*}
\]

Equivalent to:

\[
\begin{align*}
  r([], X, X). \\
  r([X|Xs], Z1, Z3) & : - r(Xs, Z1, [X|Z3]). \\
\end{align*}
\]

?- phrase(r([1,2,3,4]), L).
  L=[4,3,2,1]

A way to reverse a list in \textit{polynomial time}!
Unification

- Operation done to “match” the goal atom with the head of a clause in the program.
- Forms the basis for the matching operation we used for Prolog evaluation.
  - \( f(a, Y) \) and \( f(X, b) \) unify when \( X=a \) and \( Y=b \).
  - \( f(a, X) \) and \( f(X, b) \) do not unify.
  - \( X \) and \( f(X) \) do not unify (but they “match” in Prolog!)

Substitutions

A substitution is a mapping between variables and values (terms).
- Denoted by \( \{X_1 \mapsto t_1, X_2 \mapsto t_2, \ldots, X_n \mapsto t_n\} \) such that
  - \( X_i \neq t_i \), and
  - \( X_i \) and \( X_j \) are distinct variables when \( i \neq j \).
- Empty substitution is denoted by \( \epsilon \).
- A substitution is said to be a renaming if it is of the form \( \{X_1 \mapsto Y_1, \ldots, X_n \mapsto Y_n\} \) and \( Y_1, \ldots, Y_n \) is a permutation of \( X_1, \ldots, X_n \).
- Example: \( \{X \mapsto Y, Y \mapsto X\} \) is a renaming substitution.
Substitutions and Terms

- Application of a substitution:
  - $X\theta = t$ if $X \mapsto t \in \theta$.
  - $X\theta = X$ if $X \mapsto t \notin \theta$ for any term $t$.

- Application of a substitution $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ to a term $s$:
  - is a term obtained by \textit{simultaneously} replacing every occurrence of $X_i$ in $s$ by $t_i$.
  - Denoted by $s\theta$ and $s\theta$ is said to be an instance of $s$.

Example:

$$p(f(X, Z), f(Y, a)) \{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\} = p(f(g(Y), a), f(Z, a))$$

Composition of Substitutions

- Composition of substitutions $\theta = \{X_1 \mapsto s_1, \ldots, X_m \mapsto s_m\}$ and $\sigma = \{Y_1 \mapsto t_1, \ldots, Y_n \mapsto t_n\}$:
  - First form the set $\{X_1 \mapsto s_1\sigma, \ldots, X_m \mapsto s_m\sigma, Y_1 \mapsto t_1, \ldots, Y_n \mapsto t_n\}$
  - Remove from the set $X_i \mapsto s_i\sigma$ if $s_i\sigma = X_i$
  - Remove from the set $Y_j \mapsto t_j$ if $Y_j$ is identical to some variable $X_i$

Example: Let $\theta = \sigma = \{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\}$. Then $\theta\sigma =$

$$\{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\}\{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\} = \{X \mapsto g(Z), Y \mapsto a, Z \mapsto a\}$$

More examples: Let $\theta = \{X \mapsto f(Y)\}$ and $\sigma = \{Y \mapsto a\}$

- $\theta\sigma = \{X \mapsto f(a), Y \mapsto a\}$
- $\theta\sigma = \{X \mapsto f(Y), Y \mapsto a\}$

Composition is not \textit{commutative} but is \textit{associative}: $\theta(\sigma\gamma) = (\theta\sigma)\gamma$

Also, $E(\theta\sigma) = (E\theta)\sigma$
Unification

Idempotence

- A substitution $\theta$ is **idempotent** iff $\theta \theta = \theta$.
- Examples:
  - $\{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\}$ is not idempotent since
    
    $\{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\} \{X \mapsto g(Y), Y \mapsto Z, Z \mapsto a\} = \{X \mapsto g(Z), Y \mapsto a, Z \mapsto a\}$
  - $\{X \mapsto g(Z), Y \mapsto a, Z \mapsto a\}$ is not idempotent either since
    
    $\{X \mapsto g(Z), Y \mapsto a, Z \mapsto a\} \{X \mapsto g(Z), Y \mapsto a, Z \mapsto a\} = \{X \mapsto g(a), Y \mapsto a, Z \mapsto a\}$
  - $\{X \mapsto g(a), Y \mapsto a, Z \mapsto a\}$ is idempotent
- For a substitution $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$,
  - $\text{Dom}(\theta) = \{X_1, X_2, \ldots, X_n\}$
  - $\text{Range}(\theta) = \text{set of all variables in } t_1, \ldots, t_n$
- A substitution $\theta$ is idempotent iff $\text{Dom}(\theta) \cap \text{Range}(\theta) = \emptyset$

Unifiers

- A substitution $\theta$ is a **unifier** of two terms $s$ and $t$ if $s \theta$ is identical to $t \theta$.
- $\theta$ is a unifier of a set of equations $\{s_1 \equiv t_1, \ldots, s_n \equiv t_n\}$, if for all $i$, $s_i \theta = t_i \theta$.
- A substitution $\theta$ is more general than $\sigma$ (written as $\theta \supseteq \sigma$) if there is a substitution $\omega$ such that $\sigma = \theta \omega$
- A substitution $\theta$ is a **most general unifier** (mgu) of two terms (or a set of equations) if for every unifier $\sigma$ of the two terms (or equations) $\theta \supseteq \sigma$
- Example: Consider two terms $f(g(X), Y, a, b)$ and $f(Z, W, X, b)$.
  - $\theta_1 = \{X \mapsto a, Y \mapsto b, Z \mapsto g(a), W \mapsto b\}$ is a unifier
  - $\theta_2 = \{X \mapsto a, Y \mapsto W, Z \mapsto g(a)\}$ is also a unifier
  - $\theta_2$ is a most general unifier
Equations and Unifiers

- A set of equations $\mathcal{E}$ is in \textbf{solved form} if it is of the form
  \[ \{ X_1 \doteq t_1, \ldots, X_n \doteq t_n \} \text{ iff all } X_i \text{'s are distinct, and no } X_i \text{ appears in any } t_j. \]

- Given a set of equations in solved form $\mathcal{E} = \{ X_1 \doteq t_1, \ldots, X_n \doteq t_n \}$
  the substitution $\{ X_1/t_1, \ldots, X_n/t_n \}$ is an idempotent mgu of $\mathcal{E}$.

- Two sets of equations $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be \textbf{equivalent} if they have the same set of unifiers.

- To find the mgu of two terms $s$ and $t$, find a set of equations in solved form that is equivalent to $\{ s \doteq t \}$.
  If there is no equivalent solved form, there is no mgu.

A Simple Unification Algorithm (via Examples)

- Example 1: Find the mgu of $f(X, g(Y))$ and $f(g(Z), Z)$

  \[
  \{ f(X, g(Y)) \doteq f(g(Z), Z) \} \Rightarrow \{ X \doteq g(Z), g(Y) \doteq Z \}
  \Rightarrow \{ X \doteq g(Z), Z \doteq g(Y) \}
  \Rightarrow \{ X \doteq g(g(Y)), Z \doteq g(Y) \}
  \]

- Example 2: Find the mgu of $f(X, g(X), b)$ and $f(a, g(Z), Z)$

  \[
  \{ f(X, g(X), b) \doteq f(a, g(Z), Z) \} \Rightarrow \{ X \doteq a, g(X) \doteq g(Z), b \doteq Z \}
  \Rightarrow \{ X \doteq a, g(a) \doteq g(Z), b \doteq Z \}
  \Rightarrow \{ X \doteq a, a \doteq Z, b \doteq Z \}
  \Rightarrow \{ X \doteq a, Z \doteq a, b \doteq a \}
  \Rightarrow \text{fail}
  \]
A Simple Unification Algorithm

Given a set of equations $\mathcal{E}$:

repeat
  select $s \cdot t \in \mathcal{E}$;
  case $s \cdot t$ of
  1. $f(s_1, \ldots, s_n) \cdot f(t_1, \ldots, t_n)$:
     replace the equation by $s_i = t_i$ for all $i$
  2. $f(s_1, \ldots, s_n) \cdot g(t_1, \ldots, t_m)$, $f \neq g$ or $n \neq m$:
     halt with failure
  3. $X \cdot X$:
     remove the equation
  4. $t \cdot X$:
     where $t$ is not a variable
     replace equation by $X = t$
  5. $X \cdot t$:
     where $X \neq t$ and $X$ occurs more than once in $\mathcal{E}$:
     if $X$ is a proper subterm of $t$
       then halt with failure (5a)
     else replace all other $X$ in $\mathcal{E}$ by $t$ (5b)
until no action is possible for any equation in $\mathcal{E}$
return $\mathcal{E}$

Example 1: Find the mgu of $f(X, g(Y))$ and $f(g(Z), Z)$

\[
\{f(X, g(Y)) \cdot f(g(Z), Z)\} \Rightarrow \{X = g(Z), g(Y) \cdot Z\} \quad \text{case 1}
\Rightarrow \{X = g(Z), Z = g(Y)\} \quad \text{case 4}
\Rightarrow \{X = g(g(Y)), Z = g(Y)\} \quad \text{case 5b}
\]

Example 3: Find the mgu of $f(X, g(X))$ and $f(Z, Z)$

\[
\{f(X, g(X)) \cdot f(Z, Z)\} \Rightarrow \{X = Z, g(X) \cdot Z\} \quad \text{case 1}
\Rightarrow \{X = Z, g(Z) \cdot Z\} \quad \text{case 5b}
\Rightarrow \{X = Z, Z = g(Z)\} \quad \text{case 4}
\Rightarrow \text{fail} \quad \text{case 5a}
\]
Complexity of the unification algorithm

Consider

\[ E = \{ g(X_1, \ldots, X_n) = g(f(X_0, X_0), f(X_1, X_1), \ldots, f(X_{n-1}, X_{n-1}) \} \]

- By applying case 1 of the algorithm, we get
  \[ \{ X_1 = f(X_0, X_0), X_2 = f(X_1, X_1), \ldots, X_n = f(X_{n-1}, X_{n-1}) \} \]

- If terms are kept as trees, the final value for \( X_n \) is a tree of size \( O(2^n) \).
- Recall that for case 5 we need to first check if a variable appears in a term, and this could now take \( O(2^n) \) time.
- There are linear-time unification algorithms that share structures (terms as DAGs).
- \( X = t \) is the most common case for unification in Prolog. The fastest algorithms are linear in \( t \).
- Prolog cuts corners by omitting case 5a (the occur check), thereby doing \( X = t \) in constant time.

Most General Unifiers

- Note that mgu stands for a most general unifier.
- There may be more than one mgu. E.g. \( f(X) = f(Y) \) has two mgus:
  - \( \{ X \mapsto Y \} \)
  - \( \{ Y \mapsto X \} \)
- If \( \theta \) is an mgu of \( s \) and \( t \), and \( \omega \) is a renaming, then \( \theta \omega \) is an mgu of \( s \) and \( t \).
- If \( \theta \) and \( \sigma \) are mgus of \( s \) and \( t \), then there is a renaming \( \omega \) such that \( \theta = \sigma \omega \).