**Data Structures • Function Terms**

Data Structures are actually just Prolog Function Terms.  

Prolog Function terms do not have values. They just act like data structures.

Acknowledgements to Tony John for the Function Symbol slides that follow on function.

---

**Function Symbols in Prolog**

In logic, there are two kinds of objects: predicates and functions.

- Predicates represent statements about the world.
  - `name(A,B)` is a fact.
  - `name(A,B)` is a predicate symbol.
- Functions represent objects in the world.
  - `father(O)` is a function term.

Function terms do not have values. In Prolog, they act as data structures:

- `p3(X,Y)` denotes a point in 2-dim space.
- `p3(X,Y,Z)` denotes a point in 3-dim space.

Write a Prolog program, `distance(P1, P2, D)`, that returns the square of the distance between two points. The program should work for 2-dim and 3-dim points.

**Example:**

**Prolog Program:**

(1) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
D is X1*X1 + Y1*Y1.

(2) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
D is X1*X1 + Y1*Y1 + Z1*Z1.

**Query:**  
`distance([1,2], [3,4], D)`  
D is 5.

**Prolog Program:**

(1) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
Z1 is Z1,  
Z2 is Z2,  
D is X1*X1 + Y1*Y1 + Z1*Z1.

(2) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
Z1 is Z1,  
Z2 is Z2,  
D is X1*X1 + Y1*Y1 + Z1*Z2.

**Query:**  
`distance([1,2], [3,4], D)`  
D is 5.

**Prolog Program:**

(1) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
D is X1*X1 + Y1*Y1 + Z1*Z2.

(2) `distance(P1, P2, D)`  
`:-- X1 is X1,  
Y1 is Y1,  
X2 is X2,  
Y2 is Y2,  
D is X1*X1 + Y1*Y1 + Z1*Z2.

**Query:**  
`distance([1,2], [3,4], D)`  
D is 5.

Note: the query does not unify with any rule, so Prolog simply returns `no`, i.e., no answers for D.

---

**Returning Function Terms as Answers**

Consider, given a point, p2(X,Y), return a new point with double the coordinates. e.g.,

**Query:**  
`double(p2(3,4), P)`  
**Answer:**  
`P = p2(6,8),`  
**Sample Execution:**

**Prolog Program:**

(1) `double(p2(X,Y), P)`  
`:-- X is 2*X,  
Y is 2*Y,  
P = p2(2*X, 2*Y).

(2) `double(p2(X,Y), P)`  
`:-- X is 2*X,  
Y is 2*Y,  
P = p2(2*X, 2*Y).

**Query:**  
`double(p2(3,4), P)`  
**Answer:**  
`P = p2(6,8),`  
The body of the rule then evaluates:

**Example:**

`X is 2*X, i.e., 6`  
`Y is 2*Y, i.e., 8`  
The mgd becomes `{x|3, y|4, p|p2(6,8)}`,  
So, the answer is `P = p2(6,8),`
Recursion with Function Symbols

Example: Electrical circuits

- Two resistors in series, with resistances $R_1$ and $R_2$, respectively.
- Total resistance of the circuit is $R_1 + R_2$.
- Can represent the circuit as a function term: $\text{series}(R_1, R_2)$.

More Complex Circuits

- Two resistors in parallel, $R_1$ and $R_2$.
- Total resistance of the circuit is $\frac{1}{R_1} + \frac{1}{R_2}$.
- Represent the circuit as a function term: $\text{par}(R_1, R_2)$.

Problem:
Write a Prolog program that computes the total resistance of any circuit.

For example,

Query: $\text{resistance(series(1, 2), R)}$
Answer: $R = 1 + 2 = 3$

Query: $\text{resistance(par(2, 3), R)}$
Answer: $R = \frac{2 + 3}{2 + 3} = 6/5 = 1.2$

Query: $\text{resistance(series(3, par(2, 3)), R)}$
Answer: $R = 3 + 1.2 = 4.2$

Query: $\text{resistance(3, R)}$
Answer: $R = 3$

Solution

(1) $\text{resistance}(R, R) :- \text{number}(R)$.

(2) $\text{resistance}(\text{series}(R_1, R_2), R) :- \text{resistance}(R_1, R_3), \text{resistance}(R_2, R_4), R = R_3 + R_4$.

(3) $\text{resistance}(\text{par}(R_1, R_2), R) :- \text{resistance}(R_1, R_3), \text{resistance}(R_2, R_4), R = R_3 * R_4 / (R_3 + R_4)$.

Sample Query: $\text{resistance}(\text{series}(3, \text{par}(6, 3)), R)$

i.e., compute the total resistance, $R$, of the following circuit:

Unification

Two atomic formulas with distinct variables unify if and only if they can be made syntactically identical by replacing their variables by other terms. For example:

- $\text{loves}(bob, y)$ unifies with $\text{loves}(bob, sue)$ by replacing $y$ by $sue$.
- $\text{loves}(bob, Y)$ unifies with $\text{loves}(X, smilla)$ by replacing $Y$ by $smilla$ and $X$ by $bob$.

Both formulas become $\text{loves}(bob, smilla)$.

Formally, we use the substitution $\{ Y := \text{smilla}, X := \text{bob} \}$

which is called a unifier of $\text{loves}(bob, Y)$ and $\text{loves}(X, smilla)$.

Note that $\text{loves}(bob, X)$ does not unify with $\text{loves}(any, Y)$, since no substitution for $X$, $Y$ can make the two formulae syntactically equal.

Rules of Unification

A constant unifies only with itself.

Two structures unify if they have the same name, number of arguments, and all the arguments unify.

A variable unifies with anything. If the other thing has a value, the variable is instantiated. Otherwise, the two are associated in a way such that if one gets a value so does the other.

Unification requires all instances of the same variable in a rule to get the same value.

All rules searched, if requested by successive typing of ":"
Unification (cont.)

Examples:

p(X,X) unifies with p(b,b) and with p(c,c), but not with p(b,c),
p(X,b) unifies with p(Y,Y) with unifier X\rightarrow b, Y\rightarrow b to become p(b,b),
p(X,Z) unifies with p(Y,Y,b) with unifier X\rightarrow b, Y\rightarrow b, Z\rightarrow b to become p(b,b,b),
p(X,b,X) does not unify with p(Y,Y,c),

Prolog uses unification to compute its answer:

\text{e.g., Given the database:}

\begin{align*}
\text{oms(john, car(red, corvette))} \\
\text{oms(john, cat(black, simone, sylvester))} \\
\text{oms(elvis, copyright(song, 'jailhouse rock'))} \\
\text{oms(tolstoy, copyright(book, 'war and peace'))} \\
\text{oms(elvis, car(red, cadillac))}
\end{align*}

the query \text{oms('who', car(red, Make))} unifies with the following database facts:

\begin{itemize}
\item \text{oms(elvis, car(red, cadillac)), with unifier ['who':elvis, Make:cadillac]}
\item \text{oms(john, car(red, corvette)), with unifier ['who':john, Make:corvette]}
\end{itemize}

\textbf{Abstract Examples}

\begin{itemize}
\item \text{p(f(x),x)} unifies with \text{p(y,y)} with unifier \{x\rightarrow y, y\rightarrow x\}
to become \text{p(f(y),y)},
\item \text{p(b,f(x),y)} unifies with \text{p(b,f(y),y)} with unifier \{x\rightarrow y, y\rightarrow y, c\rightarrow v\}
to become \text{p(b,f(y),y)},
\end{itemize}

\textbf{A Negative Example}

\text{p(b,f(x),c)} does not unify with \text{p(b,f(y),v)},

\text{Reason:}

\begin{itemize}
\item To make the first arguments equal, we must replace \text{v} by \text{b},
\item To make the third arguments equal, we must replace \text{y} by \text{c},
\item These substitutions convert \text{p(b,f(u,v),v)} into \text{p(b,f(b,c),c)},
\item However, no substitution for \text{x} will convert \text{p(b,f(x),c)} into \text{p(b,f(b,c),c)},
\end{itemize}

\textbf{Another Kind of Negative Example}

\text{p(f(x),x)} does not unify with \text{p(y,y)},

\text{Reason:}

\begin{itemize}
\item Any unification would require that \text{f(x)} = \text{y} and \text{y} = \text{x},
\item But no substitution can make \text{f(x)} = \text{y},
\item \text{For example,}
\begin{itemize}
\item \text{f(a)} \neq \text{a}, using \{x\rightarrow a\}
\item \text{f(b)} \neq \text{b}, using \{x\rightarrow b\}
\item \text{f(g(a))} \neq \text{g(a)}, using \{x\rightarrow g(a)\}
\item \text{f(f(z))} \neq \text{f(z)}, using \{x\rightarrow f(z)\}
\end{itemize}
\end{itemize}

\textbf{Most General Unifiers (MGU)}

The atomic formulas \text{p(x,y)} and \text{p(g(x),v)} have infinitely many unifiers, \text{e.g.,}

\begin{itemize}
\item \{x\rightarrow g(a), y\rightarrow b, v\rightarrow g(b)\}
unifies them to give \text{p(g(a),f(b))},
\item \{x\rightarrow g(c), y\rightarrow d, v\rightarrow g(d)\}
unifies them to give \text{p(c,f(d))},
\end{itemize}

However, these unifiers are more specific than necessary.

The most general unifier (MGU) is \{x\rightarrow g(x), y\rightarrow f(x)\}
It unifies the two atomic formulas to give \text{p(g(x),f(y))}

Every other unifier results in an atomic formula of this form.

The MGU uses variables to fill in as few details as possible.

\textbf{More MGU Examples}

\begin{itemize}
\item \text{f(W, \#Z, Z)}
\item \text{f(W, X, Y, h(x))}
\item To unify these two formulas we need
\begin{align*}
Y &= g(Z) \\
Z &= h(X) \\
X &= \text{W}
\end{align*}
\item Working backwards from \text{W}, we get
\begin{align*}
Y &= g(Z) = g(h(W)) \\
Z &= h(X) = h(W) \\
X &= \text{W}
\end{align*}
\item So, the MGU is \{x\rightarrow W, y\rightarrow g(h(W)), z\rightarrow h(W)\}
\end{itemize}
Syntax of Substitutions

Formally, a substitution is a set
\[ \{ x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \} \]

where the \( x_i \)'s are distinct variable names and the \( a_i \)'s are terms that do not use any of the \( x_j \)'s.

Positive Examples:
\[ \{ x \mapsto a, y \mapsto b, z \mapsto f(x, \theta) \} \]
\[ \{ x \mapsto W, y \mapsto f(W, Y, a), z \mapsto W \} \]

Negative Examples:
\[ \{ f(x) \mapsto a \} \]
\[ \{ x \mapsto a, X \mapsto b \} \]
\[ \{ X \mapsto f(X) \} \]
\[ \{ X \mapsto f(Y), Y \mapsto a \} \]

Execution of Prolog Programs

- **Unification**: (variable bindings)
  Specializes general rules to apply to a specific problem.

- **Backward Chaining**/
  **Top-Down Reasoning**/
  **Goal-Directed Reasoning**:
  Reduces a goal to one or more subgoals.

- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backtracking.

Reasoning

- **Bottom-up** (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.
  \[ \text{e.g.}, \text{Suppose the fact } B \text{ and the rule } A \leftarrow B \text{ are given. Then infer that } A \text{ is true.} \]

- **Top-down** (or backward) reasoning: starting from the query: apply the rules in reverse, attempting only those lines of inference that are relevant to the query.
  \[ \text{e.g.}, \text{Suppose the query is } A \text{, and the rule } A \leftarrow B \text{ is given. Then to prove } A \text{, try to prove } B. \]

Top-Down Inference

A rule base:
\[ A \leftarrow B \]
\[ B \leftarrow C \]
\[ C \]

A top-down proof:
\[ \text{goal } A \]
\[ \text{goal } B \]
\[ \text{goal } C \]
\[ \text{success} \]

So, \( A \) is proved.

Top-down vs Bottom-up Inference

- **Prolog** uses top-down inference, although some other logic programming systems use bottom-up inference (e.g., Codd).
- Each has its own advantages and disadvantages:
  - Bottom-up may generate many irrelevant facts;
  - Top-down may explore many lines of reasoning that fail;
- Top-down and bottom-up inference are logically equivalent, i.e., they both prove the same set of facts.

Example 1

**Example 2**

Bottom-up inference can derive infinitely many facts.

\[ \text{Rule base:} \]
\[ p(f(x)), p(x) \]

\[ \text{Derived facts:} \]
\[ p(f(f(a))) \]

\[ \text{...} \]

Is x less than y? What is x's telephone number?
Example 3
Top-down inference may fail.

Rule base:
\[ A \Rightarrow B \]  (1)
\[ B \Rightarrow C \]  (2)

Failed line of inference:

\[ \text{goal A} \]
\[ \quad \text{rule (1)} \]
\[ \text{goal B} \]
\[ \quad \text{rule (2)} \]
\[ \text{goal C} \]
\[ \quad \text{rule (3)} \]
\[ \text{fail} \] (no rules infer C)

So, A is not proved.

Observation
Changing the order of rules and/or rule premises can cause problems for Prolog. Example:

(1) \( \text{above}(X,Y) : \text{above}(Y,Z), \text{on}(X,Y) \).
(2) \( \text{above}(X,Y) : \text{on}(X,Y) \).

Because Prolog processes premises from left to right, and rules from first to last, rule (1) causes an infinite loop.