Data Structure: the "Structure"

Simple grammar:

```prolog
<op> := <op> <op> | <op> <op> | <op> <op> | <op> <op>
<op> := { | ( | ) | : | } | <sym>
<sym> := <sym> <sym> | <sym> <sym> | <sym> <sym>
<sym> := <sym> | <sym> | <sym> | <sym> | <sym>
```

Parse tree:

Representation as a Prolog structure:

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

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

Write a Prolog program, `SQUAD(Point1,Point2,D),` that returns the square of the distance between two points. The program should work for 2- and 3-dim points.

Want:

```prolog
SQUAD(p3(1,2), p3(3,6), D).
returns D = (3-1)**2 + (6-2)**2 = 4**2 + 4**2 = 16
```

and

```prolog
SQUAD(p3(1,0), p3(2,2), D).
returns D = (2-1)**2 + (2-0)**2 = 1**2 + 2**2 = 5
```

and

```prolog
SQUAD(p3(0,0), p3(1,1), D).
is undefined
```

Prolog Program:

```
(1) SQUAD(p3(X1,Y1), p3(X2,Y2), D).
:: XD is X1-X2,
   YD is Y1-Y2,
   D is XD*XD + YD*YD,

(2) SQUAD(p3(X1,Y1,Z1), p3(X2,Y2,Z2), D).
:: XD is X1-X2,
   YD is Y1-Y2,
   ZD is Z1-Z2,
   D is XD*XD + YD*YD + ZD*ZD,
```

Query: `SQUAD(p3(1,2), p3(3,6), D).

This query unifies with the head of rule (1) with \[x1=1, y1=2, x2=3, y2=6\]

so.

```prolog
XD is X1-X2 = 2
YD is Y1-Y2 = 4
D is X^2 + Y^2 = 4 + 16 = 20
```

Note: the query does not unify with the head of rule (2), so only rule (1) is used.

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 Banerji for the Function Symbol slides that follow on functions.

Function Symbols in Prolog

In logic, there are two kinds of objects: 

- **predicates** and 
  **functions**

  - Predicates represent **statements** about the world:
    - John hates Mary: `hates(john,mary),` 
    - Jane is short: `short(jane)` 
    (hates is a predicate symbol, short(jane) is an atomic formula)
  
  - Function terms represent objects in the world: 
    - the mother of Mary: `mother_of(mary)`
    - a rectangle of length 3 and width 4: `rectangle(3,4)`
    (mother_of(mary) is a function term, rectangle is a function symbol)

Prolog Program:

```
(1) SQUAD(p3(X1,Y1), p3(X2,Y2), D).
:: XD is X1-X2,
   YD is Y1-Y2,
   D is XD*XD + YD*YD,

(2) SQUAD(p3(X1,Y1,Z1), p3(X2,Y2,Z2), D).
:: XD is X1-X2,
   YD is Y1-Y2,
   ZD is Z1-Z2,
   D is XD*XD + YD*YD + ZD*ZD,
```

Query: `SQUAD(p3(1,0), p3(2,2), D).

This query unifies with the head of rule (2), with \[x1=1, y1=0, x2=2, y2=2, z1=0\]

so.

```prolog
XD is X1-X2 = 1
YD is Y1-Y2 = 2
ZD is Z1-Z2 = 2
D is X^2 + Y^2 + Z^2 = 1 + 4 + 4 = 9
```

Note: this query does not unify with any rule, so Prolog simply returns `no,` i.e., no answers for D.
Returning Function Terms as Answers

Given a point, \( p_2(x,y) \), return a new point with double the coordinates, e.g.,

**Query:** \( \text{double}(p(3,4), p) \)

**Answer:** \( p = p(6,8) \).

**Prolog Program:**

\[
\text{double}(p_1(x_1,y_1), p_2(x_2,y_2)) \implies \begin{cases} x_2 &= 2x_1, \\ y_2 &= 2y_1. \end{cases}
\]

**Query:** \( \text{double}(p(3,4), p) \)

The query unifies with the head of the rule, where the mgu is

\[ \{x_1:3, y_1:4, P:p(6,8)\} \]

The body of the rule then evaluates:

- \( x_2 = 2x_1 \), i.e., 6
- \( y_2 = 2y_1 \), i.e., 8

The mgu becomes \( \{x_1:3, y_1:4, P:p(6,8)\} \).

So, the answer is \( P = p(6,8) \).

Sample Execution

**Prolog Program:**

\[
\text{double}(p_1(x_1,y_1), p_2(x_2,y_2)) \implies \begin{cases} x_2 &= 2x_1, \\ y_2 &= 2y_1. \end{cases}
\]

**Query:** \( \text{double}(p(3,4), p) \)

In Plain English: if \( x_2 = 2x_1 \) and \( y_2 = 2y_1 \), then the double of \( p_1(x_1,y_1) \) is \( p_2(x_2,y_2) \).

An equivalent program using "w":

\[
\text{double}(p_1(x_1,y_1), P) \implies x_2 = 2x_1, y_2 = 2y_1, P = p_2(x_2,y_2).
\]

Here, "w" is being used to assign a value to variable \( P \). Try to avoid this!!!! It reflects procedural thinking.

**Problem:**

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

**For example,**

**Query:** \( \text{resistance}(\text{series}(1,2), R) \)

**Answer:** \( R = 1 + 2 = 3 \)

**Query:** \( \text{resistance}(\text{par}(2,3), R) \)

**Answer:** \( R = \frac{2\times3}{2+3} = 6/5 = 1.2 \)

**Query:** \( \text{resistance}(\text{series}(3,\text{par}(2,3)), R) \)

**Answer:** \( R = 3 + 1.2 = 4.2 \)

**Query:** \( \text{resistance}(3, R) \)

**Answer:** \( R = 3 \)

**Solution**

1. \( \text{resistance}(\text{par}(R_1, R_2), R) : \text{number}(R) \)
2. \( \text{resistance}(\text{series}(C_1, C_2), R) : \text{number}(R) \)
   - \( R = \frac{1}{\frac{1}{C_1} + \frac{1}{C_2}} \)
3. \( \text{resistance}(\text{par}(C_1, C_2), R) : \text{number}(R) \)
   - \( R = \frac{1}{\frac{1}{C_1} + \frac{1}{C_2}} \)

**Sample Query:**

\( \text{resistance}(\text{series}(3,\text{par}(6,3)), R) \)

\( \text{i.e., compute the total resistance, } R, \text{ of the following circuit:} \)

**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 \( R_1 + R_2 \), respectively.
- Can represent the circuit as a function term:

\[
\text{series}(R_1, R_2)
\]

- Two resistors in **parallel**, with total resistance of the circuit is \( \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}} \), respectively.
- Can represent the circuit as a function term:

\[
\text{par}(R_1, R_2)
\]

**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 all possible solutions that can be obtained via unification and backchaining.
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:
- \( \text{loves}(\text{bob}, Y) \) unifies with \( \text{loves}(\text{bob}, \text{santa}) \) by replacing \( Y \) by \( \text{santa} \).
- \( \text{loves}(\text{bob}, Y) \) unifies with \( \text{loves}(X, \text{santa}) \) by replacing \( X \) by \( \text{bob} \).

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

Formally, we use the substitution

\[ \{Y\rightarrow\text{santa}, X\rightarrow\text{bob}\} \]

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

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

**Rules of Unification**

A constant unifies only with itself.

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

A variable unifies with anything. If the other thing is 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 with Function Terms

Prolog uses unification to compute its answers, e.g., Given the database:

- \( \text{owns}((\text{john}, \text{car}(\text{red}, \text{corvette}))) \)
- \( \text{owns}((\text{john}, \text{cat}(\text{black}, \text{simone}, \text{sylvester}))) \)
- \( \text{owns}((\text{elvis}, \text{copyright}(\text{song}, \text{"jailhouse rock"}))) \)
- \( \text{owns}((\text{coltspoy}, \text{copyright}(\text{book}, \text{"war and peace"}))) \)
- \( \text{owns}((\text{elvis, car}(\text{red}, \text{cadillac}))) \)

The query \( \text{owns}((\text{who, car}(\text{red}, \text{make}))) \) unifies with the following database facts:

- \( \text{owns}((\text{elvis, car}(\text{red}, \text{cadillac}))) \)
- \( \text{owns}((\text{john, car}(\text{red}, \text{corvette}))) \)

**Most General Unifiers (MGU)**

The atomic formulas \( p(x, f(y)) \) and \( p(g(u), v) \) have infinitely many unifiers, e.g.,

- \( \{x\rightarrow g(u), y\rightarrow v, u\rightarrow v, f(y)\rightarrow f(u)\} \)
- \( \{x\rightarrow g(u), y\rightarrow v, u\rightarrow v, f(y)\rightarrow f(u)\} \)

However, these unifiers are more specific than necessary.

The most general unifier (MGU) is

\( \{x\rightarrow g(u), y\rightarrow v(\text{d}(y))\} \)

It unifies the two atomic formulas to give \( p(g(u), v(\text{d}(y))) \).

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

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

\[ f(W, \not\exists Z, Z) \]
\[ f(X, Y, h(X)) \]

To unify these two formulas, we need

\[
\begin{align*}
Y &= g(Z) \\
Z &= h(X) \\
X &= W
\end{align*}
\]

Working backwards from \( W \), we get

\[
\begin{align*}
Y &= g(Z) = g(h(W)) \\
Z &= h(X) = h(W) \\
X &= W
\end{align*}
\]

So, the mgu is

\[ \{ X/W, Y/g(h(W)), Z/h(W) \} \]

**More MGU Examples**

<table>
<thead>
<tr>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>MGU</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x, a) )</td>
<td>( f(g, Y) )</td>
<td></td>
</tr>
<tr>
<td>( f(h(x, b)) )</td>
<td>( f(h(g(x, b), y), b) )</td>
<td></td>
</tr>
<tr>
<td>( g(f, h(x)) )</td>
<td>( g(y, f(Y, Z), Z) )</td>
<td></td>
</tr>
<tr>
<td>( f(x, g(x, 2)) )</td>
<td>( f(z, y, h(y)) )</td>
<td></td>
</tr>
<tr>
<td>( f(x, y, h(2)) )</td>
<td>( f(g(y, z), h(g(Q, Z))) )</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax of Substitutions**

Formally, a substitution is a set

\[ \{ y_1 \rightarrow t_1, \ldots, y_n \rightarrow t_n \} \]

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

<table>
<thead>
<tr>
<th>Positive Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ { x/y } ]</td>
</tr>
<tr>
<td>[ { x/y, x/b } ]</td>
</tr>
<tr>
<td>[ { x/f(x) } ]</td>
</tr>
<tr>
<td>[ { x/Y, y/\alpha } ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Negative Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ { f(x) } ]</td>
</tr>
<tr>
<td>[ { x/y, x/b } ]</td>
</tr>
<tr>
<td>[ { x/f(x) } ]</td>
</tr>
<tr>
<td>[ { x/Y, y/\alpha } ]</td>
</tr>
</tbody>
</table>