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 Bonner 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).
John is short:  short(john)
```

(`hates` is a predicate symbol, `short(john)` 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)

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

let `p2(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, `SQDIST(Point1,Point2,D)`, that returns the square of the distance between two points. The program should work for 2- and 3-dim points.

Want:

```
SQDIST(p2(1,2), p2(3,5), D)
  returns D = (3-1)**2 + (5-2)**2
            = 4+9 = 13
and
SQDIST(p3(1,1,0), p3(2,2,3), D)
  returns D = (1-2)**2 + (1-2)**2 + (0-3)**2
            = 1+1+9 = 11
and
SQDIST(p2(0,0), p3(1,1,1), D)
  is undefined
```

Prolog Program:

```
(1) SQDIST(p2(X1,Y1), p2(X2,Y2), D)
      :- XD is X1-X2,
         YD is Y1-Y2,
         D is XD*XD + YD*YD.

(2) SQDIST(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: `SQDIST(p2(1,2), p2(3,5), D)`

This query unifies with the head of rule (1) with {X1\1, Y1\2, X2\3, Y2\5}

so, XD is X1-X2 = 1-3 = -2
YD is Y1-Y2 = 2-5 = -3
D is $(-2)^2 + (-3)^2 = 13$
So, D=13 is returned

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

Prolog Program:

```
(1) SQDIST(p2(X1,Y1), p2(X2,Y2), D)
      :- XD is X1-X2,
         YD is Y1-Y2,
         D is XD*XD + YD*YD.

(2) SQDIST(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: `SQDIST(p3(1,1,0),p3(2,2,3),D).`

This query unifies with the head of rule (2), with {X1\1, Y1\1, Z1\0, X2\2, Y2\2, Z2\3}

```
so, XD is 1-2 = -1
    YD is 1-2 = -1
    ZD is 0-3 = -3
    D  is 1+1+9 = 11
```

So, D=11 is returned

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

Prolog Program:

```
(1) SQDIST(p2(X1,Y1), p2(X2,Y2), D)
      :- XD is X1-X2,
         YD is Y1-Y2,
         D is XD*XD + YD*YD.

(2) SQDIST(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: `SQDIST(p2(0,0), p3(1,1,1), D).`

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

e.g., 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).`

Prolog Program:

```
double(p2(X1,Y1), p2(X2,Y2))
   :- X2 is 2*X1,
      Y2 is 2*Y1.
```

In Plain English: if `X2 = 2*X1` and `Y2 = 2*Y1`, then the double of `p2(X1,Y1)` is `p2(X2,Y2)`.

An equivalent program using "=":

```
double(p2(X1,Y1), P)
   :- X2 is 2*X1, Y2 is 2*Y1,
      P = p2(X2,Y2).
```

Here, "=" is being used to assign a value to variable `P`. Try to avoid this!!!!! It reflects procedural thinking.

Sample Execution

Prolog Program:

```
double(p2(X1,Y1), p2(X2,Y2))
   :- X2 is 2*X1,
      Y2 is 2*Y1.
```

Query: `double(p2(3,4),P)`

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

$$\{X1\backslash 3,\ Y1\backslash 4,\ P\backslash p2(X2,Y2)\}$$

The body of the rule then evaluates:

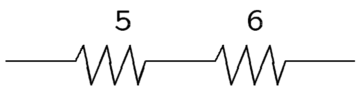
```
X2 is 2*X1,   i.e., 6
Y2 is 2*Y1,   i.e., 8
```

The mgu becomes $\{X1\backslash 3,\ Y1\backslash 4,\ P\backslash p2(6,8)\}$.

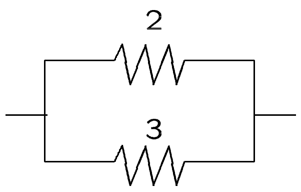
So, the answer is `P = p2(6,8).`

Recursion with Function Symbols

Example: Electrical circuits

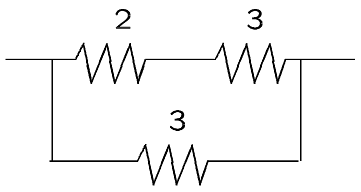


• Two resistors in series, with resistances R_1 and R_2, respectively.
• Total resistance of the circuit is $5 + 6 = 11$.
• Can represent the circuit as a function term: `series(5,6)`.

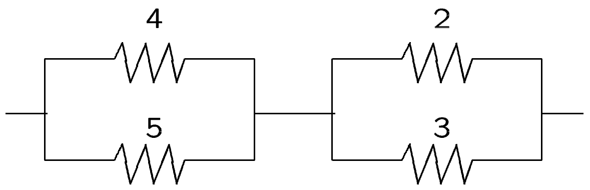


• Two resistors in parallel.
• Total resistance of the circuit is $\frac{2\times 3}{2+3} = 1.2$
• Represent the circuit as a function term: `par(2,3)`.

More Complex Circuits



`par(3, series(2,3))`



`series(par(4,5), par(2,3))`

Problem:

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

For example,

Query: `resistance(series(1,2), R)`
Answer: `R = 1+2 = 3`

Query: `resistance(par(2,3), R)`
Answer: `R = (2*3)/(2+3) = 6/5 = 1.2`

Query: `resistance(series(3,par(2,3)), R)`
Answer: `R = 3 + 1.2 = 4.2`

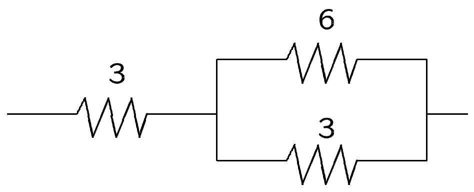
Query: `resistance(3, R)`
Answer: `R = 3`

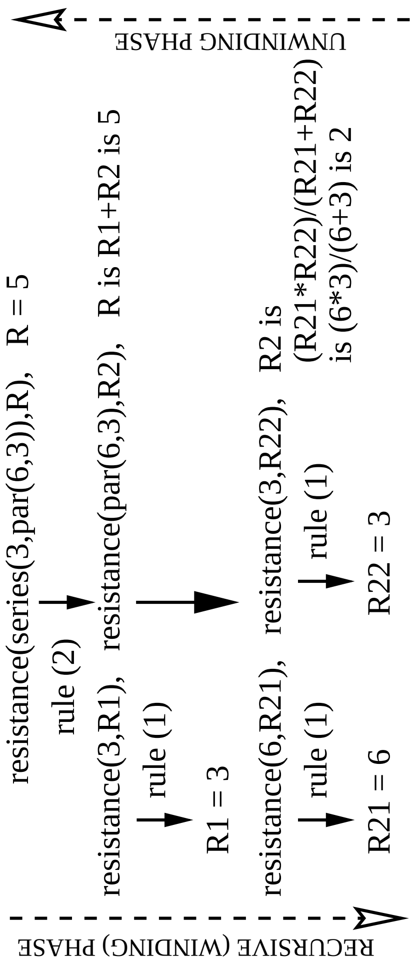
Solution

```
(1)   resistance(R,R) :- number(R).

(2)   resistance(series(C1,C2), R)
         :- resistance(C1, R1),
            resistance(C2, R2),
            R is R1+R2.

(3)   resistance(par(C1,C2), R)
         :- resistance(C1,R1),
            resistance(C2,R2),
            R is (R1*R2)/(R1+R2).
```

Sample Query:
`resistance(series(3,par(6,3)), TR)`

i.e., compute the total resistance, `TR`, of the following circuit:




resistance(series(3,par(6,3)),R), R = 5
rule (2)
resistance(3,R1), resistance(par(6,3),R2), R is R1+R2 is 5
rule (1)
R1 = 3
resistance(6,R21), resistance(3,R22), R2 is (R21*R22)/(R21+R22) is (6*3)/(6+3) is 2
rule (1)
rule (1)
R21 = 6
R22 = 3
RECURSIVE (WINDING) PHASE
UNWINDING PHASE

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 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,

- `loves(bob,Y)` unifies with `loves(bob,sue)` by replacing `Y` by `sue`.
- `loves(bob,Y)` unifies with `loves(X,santa)` by replacing `Y` by `santa` and `X` by `bob`.

  Both formulas become `loves(bob,santa)`.

  Formally, we use the **substitution**

  $$\{Y\backslash santa, X\backslash bob\}$$

  which is called a **unifier** of `loves(bob,Y)` and `loves(X,santa)`.
- Note that `loves(bob,X)` does *not* unify with `loves(tony,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 iff 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 variabe 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 b,Y b to become p(b,b).

p(X,Z,Z) unifies with P(Y,Y,b) with unifier X b,Y b,Z b to become p(b,b,b).

p(X,b,X) does not unify with p(Y,Y,c).

Unification with Function Terms

Prolog uses unification to compute its answers.

e.g., Given the database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

the query `owns(Who,car(red,Make))`
unifies with the following database facts:

- `owns(elvis,car(red,cadillac))`,
  with unifier `{Who\elvis, Make\cadillac}`
- `owns(john,car(red,corvette))`,
  with unifier `{Who\john, Make\corvette}`

Abstract Examples

- `p(f(X),X)` unifies with `p(Y,b)`
  with unifier {`X\b`, `Y\f(b)`}
  to become `p(f(b),b)`.

- `p(b,f(X,Y),c)` unifies with `p(U,f(U,V),V)`
  with unifier {`X\b`, `Y\c`, `U\b`, `V\c`}
  to become `p(b,f(b,c),c)`.

A Negative Example

`p(b,f(X,X),c)` does *not* unify with `p(U,f(U,V),V)`.

Reason:

- To make the first arguments equal,
  we *must* replace `U` by `b`.

- To make the third arguments equal,
  we *must* replace `V` by `c`.

- These substitutions convert
  `p(U,f(U,V),V)` into `p(b,f(b,c),c)`.

- However, *no* substitution for `X` will convert
  `p(b,f(X,X),c)` into `p(b,f(b,c),c)`.

Another Kind of Negative Example

`p(f(X),X)` does *not* unify with `p(Y,Y)`.

Reason:

- Any unification would require that
  $f(X) = Y$ and $Y = X$

- But no substitution can make
  $f(X) = X$

- For example,

  $f(a) \neq a$, using $\{X\backslash a\}$

  $f(b) \neq b$, using $\{X\backslash b\}$

  $f(g(a)) \neq g(a)$, using $\{X\backslash g(a)\}$

  $f(f(c)) \neq f(c)$, using $\{X\backslash f(c)\}$

  etc.

Most General Unifiers (MGU)

The atomic formulas `p(X,f(Y))` and `p(g(U),V)` have infinitely many unifiers. *e.g.*,

- $\{X\backslash g(a),\ Y\backslash b,\ U\backslash a,\ V\backslash f(b)\}$
  unifies them to give `p(g(a),f(b))`.

- $\{X\backslash g(c),\ Y\backslash d,\ U\backslash c,\ V\backslash f(d)\}$
  unifies them to give `p(g(c),f(d))`.

However, these unifiers are more specific than necessary.

The most general unifier (mgu) is

$$\{X\backslash g(U),\ V\backslash f(Y)\}$$

It unifies the two atomic formulas to give `p(g(U),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.

MGU Example

$$f(W,\ g(Z),\ Z)$$

$$f(X,\ Y,\ h(X))$$

To unify these two formulas, we need

$$\begin{array}{rcl} Y & = & g(Z) \\ Z & = & h(X) \\ X & = & W \end{array}$$

Working backwards from W, we get

$$\begin{array}{rcl} Y & = & g(Z) = g(h(W) \\ Z & = & h(X) = h(W) \\ X & = & W \end{array}$$

So, the mgu is

$$\{X\backslash W,\ Y\backslash g(h(W)),\ Z\backslash h(W)\}$$

More MGU Examples

t_1	t_2	MGU
f(X,a)	f(a,Y)	
f(h(X,a),b)	f(h(g(a,b),Y),b)	
g(a,W,h(X))	g(Y,f(Y,Z),Z)	
f(X,g(X),Z)	f(Z,Y,h(Y))	
f(X,h(b,X))	f(g(P,a),h(b,g(Q,Q)))	

Syntax of Substitutions

Formally, a substitution is a set

$$\{v_1\backslash t_1, \ldots, v_n\backslash t_n\}$$

where the v_i's are distinct variable names
and the t_i's are terms that do not use **any** of the v_j's.

Positive Examples:

$\{X\backslash a,\ Y\backslash b,\ Z\backslash f(a,b)\}$
$\{X\backslash W,\ Y\backslash f(W,V,a),\ Z\backslash W\}$

Negative Examples:

$\{f(X)\backslash a\}$
$\{X\backslash a,\ X\backslash b\}$
$\{X\backslash f(X)\}$
$\{X\backslash f(Y),\ Y\backslash g(q)\}$

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 backchaining.

Reasoning

- **Bottom-up** (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  e.g., Suppose the fact B and the rule $A \leftarrow B$ are given. Then infer that A 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.

  e.g., Suppose the query is A, and the rule $A \leftarrow B$ is given. Then to prove A, try to prove B.

Bottom-up Inference

A rule base:

```
A <- B        (1)
B <- C        (2)
C             (3)
```

A bottom-up proof:

$$\begin{tikzcd}
\text{infer A} \\
\text{infer B} \arrow[u, "\text{rule (1)}"'] \\
\text{infer C} \arrow[u, "\text{rule (2)}"'] \\
\text{start} \arrow[u, "\text{rule (3)}"']
\end{tikzcd}$$

So, A is proved

Top-Down Inference

A rule base:

```
A <- B          (1)
B <- C          (2)
C               (3)
```

A top-down proof:

$$\begin{tikzcd}
\text{goal A} \arrow[d, "\text{rule (1)}"] \\
\text{goal B} \arrow[d, "\text{rule (2)}"] \\
\text{goal C} \arrow[d, "\text{rule (3)}"] \\
\text{success}
\end{tikzcd}$$

So, A is proved

Top-down vs Bottom-up Inference

- Prolog uses top-down inferene, although some other logic programming systems use bottom-up inference (*e.g.*, Coral).
- 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

Bottom-up inference can derive
many facts.

Rule base:

```
p(X,Y,Z) <- q(X),q(Y),q(Z).
q(a1).
q(a2).
...
q(an).
```

Bottom-up inference derives n^3 facts of the form p(a$_i$,a$_j$,a$_k$):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```

Example 2

Bottom-up inference can derive
infinitely many facts.

Rule base:

```
p(f(x)) <- p(x).
p(a).
```

Derived facts:

```
p(f(a))
p(f(f(a)))
p(f(f(f(a))))
...
```

In contast, top-down inference derives only the facts requested by the user, e.g.

```
who does jane love?
what is john's telephone number?
```

Example 3

Top-down inference may fail.

Rule base:

```
A <- B   (1)
B <- C   (2)
```

Failed line of inference:

$$\begin{tikzcd}
\text{goal A} \arrow[d, "\text{rule (1)}"] \\
\text{goal B} \arrow[d, "\text{rule (2)}"] \\
\text{goal C} \arrow[d, "\text{rule (3)}"] \\
\text{fail} \\[-2em]
\text{(no rules infer C)}
\end{tikzcd}$$

So, A is not proved

Observation

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

```
(1)   above(X,Z) :- above(Y,Z), on(X,Y).
(2)   above(X,Y) :- on(X,Y).
```

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