Logic Programming: Prolog

Table of Contents

Prolog

Predicates

- logic programming language based on predicate calculus
- build on **predicates** which define **relations** among their arguments
- e.g. relationships: parent/child
- predicates can be defined by 1+ **clauses**
- **fact/unit clause** *classes.pl*:

```
% the students we know about
student(alice).
student(bob).
student(claire).
student(don).
% who is enrolled in which subjects
enrolled(alice, logic).
enrolled(alice, maths).
enrolled(bob, maths).
enrolled(claire, physics).
enrolled(don, logic).
enrolled(don, art_history).
```

You can then load this file, and make queries:

```
?- [classes].
true.
?- student(bob).
true.
?- student(sally).
false.
```

- **closed world assumption**: anything you haven't said to be true is assumed to be false

Variables

- **variables** in Prolog can only hold one value each time it exists, and refers to the same value each place it appears in that scope. Think of it as standing in for a value we don't yet know

- must begin with capital letter or underscore, containing only letters, digits, underscores If you pose queries with variables in them, Prolog looks for bindings that satisfy the query:

```
?- student(X).
X = alice ;
X = bob ;
X = claire ;
X = don.
```

The semicolon is input by the user to move to the next possible binding. Enter accepts a binding.

```
?- enrolled(alice, Subject).
Subject = logic ;
Subject = maths.
```

Special variable _ matches anything, and each place you write it, it names a different variable.

```
% is alice enrolled in any subject?
?- enrolled(claire, _).
true.
% is anyone enrolled in any subject?
?- enrolled(_, _).
true ;
true ;
...
true.
```

Compound queries

- queries can involve conjunctions (`AND`), disjunctions (`OR`), and negations (`NOT`)
- conjunction operator: `,`
- disjunction operator: `;`
- negation operator: `\+`

```
% who is taking both maths and logic?
?- enrolled(S, maths), enrolled(S, logic).
S = alice ;
false.
% who is enrolled in either maths or logic?
?- enrolled(S, maths) ; enrolled(S, logic).
S = alice ;
S = don ;
S = alice ;
S = bob.
% who is enrolled in logic but not maths?
?- enrolled(S, logic), \+ enrolled(S, maths).
S = don.
```

Rules

Facts are clauses specifying that a relationship holds. **Rules** are clauses that specifies that a relationship holds under certain conditions.

```
head :- body
```

The rule specifies that `head` holds if `body` holds

```
% general syntax: the rule specifies that the relationship holds if
head :- body
% two people are classmates if they are enrolled in the same class
classmates(X, Y) :- enrolled(X, Class), enrolled(Y, Class)
```

Equality

```
% this shows bob is his own classmate
?- classmates(bob, X).
X = alice ;
X = bob.
% we can use negation and equality to rectify:
?- classmates(X, Y) :-
    enrolled(X, Class),
    enrolled(Y, Class),
    \+ X = Y.
?- classmates(bob, X).
X = alice ;
false.
```

Disequality and Negation as Failure

- `\=`: not equal predicate.. `X \= Y` behaves the same as `\+ X = Y`
- **negation as failure**: Prolog negates a query by attempting to find a solution: if it succeeds, the negation fails. If it fails, the negation succeeds. It doesn't bind any variables, so negations should be written *following* goals that do bind variables used in the negation.

```
% this doesn't work properly:
?- X \= Y, X = bob, Y = alice.
false.
?- X = bob, Y = alice, X \= Y.
X = bob,
Y = alice.
```

Terms

- Prolog is **dynamically typed**. All data are called **terms**.
- **atomic terms**: primitive types. Integers, floating point numbers, and atoms.
- atoms can begin with a lowercase letter and follow with letters, digits, or underscores, otherwise it begins/ends with a single quote `'` and can contain any characters
- the Prolog compiler will not identify type errors in the code
- any argument of any predicate you define can have any type

Compound Terms

- compound term: Prolog equivalent to C `struct`. Begins with a *functor* (an atom) and follows with 1+ terms as arguments
- e.g. compound term with functor `card`, arity 2, first argument is clubs, second argument is 3.

```
1 card(clubs, 3)
```

Lists

- `[]` : empty list
- `[E|Es]` : non empty list, `E`: head, `Es`: tail
- e.g. `[E1,E2,E3|Es]`

Unification

- variables in Prolog are a kind of data that stands for a currently unknown value, and continue to exist after the predicate that creates them finishes executing.
- variables become bound through **unification**, which takes two terms and tries to make them identical, binding variables as necessary
- if a set of *consistent* bindings cannot be found for all variables, unification fails
- unification happens at every predicate call: the call is unified with the head of the first clause for the predicate: if it succeeds, Prolog executes the body of the clause; if the unification fails, Prolog goes to the next clause for the predicate and tries the same thing
- the equality predicate = also unifies its two arguments

`length`

Here's an implementation of Haskell's `take` that returns the first `N` elements of a list

```
take(N, List, Front) :-
    length(Front, N),
    append(Front, _, List).
```

`Front` is the first `N` elements of `List` if the length of `Front` is `N` and you can append `Front` to something to produce `List`.

`member`

- `member(E, List)` holds when `E` is one of the elements of `List`
- use this to check whether `E` is an element of `List`, or to have Prolog produce elements of `List` one at a time

`select`

- `select(Elem, List1, List2)`: `List2` contains everything in `List1` except `Elem`
  - can use to remove single occurrence of `Elem` from `List1`, or insert `Elem` in any place in `List2`, or to select an element of `List1`, producing 1 element plus the rest of the list

`nth0/3`

- `nth0(Index, List, Elem)`: finds the *n* th element of `List` (0-based).
- can determine position of `Elem` in `List`
- can produce elements of `List` together with their positions.

`nth0/4`

- `nth0(N, List, Elem, Rest)`: same as `nth0/3`, but `Rest` is the list of elements other than `Elem`
- use it to remove an element from a list by position, by value while providing position, or to insert an element at a particular position

Documenting Modes

- document each Prolog predicate in a comment before the predicate definition
- give each argument a character indicating its mode

- `+`: input argument. Expected to be bound when the predicate is called
- `-`: output argument. Normally unbound when the predicate is called. If it is bound, it will be unified with the output
- `?`: the predicate may be input/output/both
- e.g. `append/3`

```
% append(+List1, +List2, -List3)
% append(-List1, -List2, +List3)
% List3 is a list of all the elemnts of List1 in order followed
% by the all the elements of List2 in order.
```

- documentation should indicate all intended modes of use, and be clear when it doesn't work

Arithmetic

- `is/2` is used to evaluate expressions:

```
?- X is 6*7.
X=42.
```

- the 2nd argument must be a ground term!

```
?- X is 1*A.
ERROR: Arguments are not sufficiently instantiated.
ERROR: In:
...
```

- Prolog is not a symbolic computation system, with very limited ability to reason about arithmetic

Arithmetic Predicates

Predicates	Description
`V is Expr`	Unify V with the value of expression Expr
`Expr1 =:= Expr2`	Succeeds if Expr1 and Expr2 are equal
`Expr1 =\= Expr2`	Succeeds if Expr1 and Expr2 are different
`Expr1 < Expr2`	Succeeds if Expr1 is strictly less than the value of Expr2
`Expr1 =< Expr2`	Succeeds if Expr1 is less or equal to the value of Expr2

Predicates	Description
`Expr1 > Expr2`	Succeeds if Expr1 is strictly greater than the value of Expr2
`Expr1 >= Expr2`	Succeeds if Expr1 is greater or equal to the value of Expr2

Arithmetic Expressions

Function	Description
`-X`	unary negation (integer or float)
`X + Y`	addition (integer or float)
`X - Y`	subtraction (integer or float)
`X * Y`	multiplication (integer or float)
`X / Y`	division, producing integer or float
`X //Y`	integer division, rounding toward zero
`X rem Y`	integer remainder, same sign as X
`X div Y`	integer division, rounding down
`X mod Y`	integer modulus, same sign as Y
`integer(X)`	round X to the nearest integer
`float(X)`	floating point value of X
`ceil(X)`	smallest integer >= X
`floor(X)`	largest integer =< X
`max(X,Y)`	larger of X and Y (integer or float)
`min(X,Y)`	smaller of X and Y (integer or float)

Semantics

- **semantics** of a logic program: what does it make true?
- i.e. a program consisting of a set of ground facts
- to determine the semantics of a program containing rules, you start with an empty set of clauses, and then copy all facts into the semantics

- then for each `Head :- Body`, unify all goals in `Body` with every combination of facts in the semantics. Add each combination instance of `Head` to the semantics
- when this process reaches a **fixed point**, where no new clauses are added to the semantics, you are done.

Tail recursion

- make recursive calls operate more like iterative calls, using constant stack space instead of linear stack space
- this works when the last call in the predicate is recursive
- you can typically do this by adding an **accumulator** argument to the predicate that stores an intermediate result

e.g. we want to make `sumList2` tail recursive

```
sumlist([], 0).
sumlist([E|Es], N) :-
    sumlist(Es, N1),
    N is N1 + E.
```

As a for loop, this would be:

```
N = 0
for elem in list:
    N = N + elem

return N
```

To translate this to Prolog, create a new predicate `sumlist/3` with an accumulator

```
% make sumlist/2 call sumlist/3 with initialised accumulator
sumlist(List, Sum) :-
    sumlist(List, 0, Sum).

sumlist([], Sum, Sum).
sumlist([E|Es], Sum0, Sum) :-
    Sum1 is Sum0 + E,
    sumlist(Es, Sum1, Sum).
```

Determinism

- for efficiency you should make predicates deterministic (i.e. leave no choice points) when there are no other solutions
- if choicepoints remain, it disables tail recursion optimisation

Indexing

When you call a predicate with some arguments bound, Prolog looks at all the clauses for the predicate to see if some have non-variables in that position of the cluase head. If so, it constructs an index on that argument position. Then Prolog can jump straight to the first clause that matches the query. If it is forced to backtrack, it will jump to the next clause that could match. When it knows there are no more clauses that could match, it removes the choicepoint.

`if -> then ; else`

- Prolog goal of the form `(p -> q ; r)`: first calls p. If that succeeds then call `q` and ignore `r`. Otherwise ignore `q` and call `r`
- used to produce determinism
- note that if `p` has more than one solution, Prolog commits to the first solution and throws away the others
- good practice to write the `;` at the start of the line to distinguish it from the `,`

```
ints_between(N0, N, List) :-
    (  N0 < N
    -> List = [N0|List1],
       N1 is N0 + 1,
       ints_between(N1, N, List1)
    ;  N0 = N,
       List = [N]
    ).
```