Logic Programming: Prolog

Table of Contents

Prolog

Key points

- **closed world assumption:** anything you haven't said to be true is assumed to be false
- **negation as failure:** to execute `\+G`, Prolog first tries to prove `G`. If it fails, `\+G` succeeds. Otherwise, it fails.
  - failing goals does not bind variables, so `\+G` cannot solve for variables
  - ensure all variables in a negated goal are bound before the goal is executed
- **terms:** all data structures are terms. They can be
  - **atomic:**
    * integers, floating point numbers
    * **atoms:** begins with lower case letter
  - **compound:** functor/function symbol followed by 0+ arguments
  - **variable:** denotes a single unknown term, begins with upper case letter
    * single `_` specifies a different variable each time it appears
- **Datalog:** fragment of Prolog with only atomic terms and variables
- **single-assignment language:** a variable can only be bound once
- **ground term:** contains no variables
  - has only one instance
- **nonground term:** contains 1+ variable
  - has an infinite number of instances
- **substitution:** mapping from variables to terms
  - never replaces atomic/compound terms, only replaces variables
- **applying a substitution:** consistently replacing occurrences of each variable in the map with the term it is mapped to
- do arithmetic with the `is/2` predicate, infix e.g. `X is 6*7`
  - `is/2` only works if the 2nd argument is ground

Proper List

- either empty `[]` or not `[X|Y]`
- if not empty, the tail `Y` must be a proper list

```
proper_list([]).
proper_list([Head|Tail]) :- proper_list(Tail).
```

- this is a common pattern for recursive list traversal in Prolog

Append

```
append([], C, C).
append([A|B], C, [A|BC]) :- append(B, C, BC).
```

Member

```
member1(Elt, List) :- append(_, [Elt|_], List).

member2(Elt, [Elt|_]).
member2(Elt, [_|Rest]) :- member2(Elt, Rest).
```

- `member2` is more efficient: `member1` builds and ignores the list of elements before `Elt` in `List`. The second does not.

Logic and Resolution

Interpretations

- **interpretation**:
  - atomic terms stand for entities in the **domain of discourse (universe)**
  - each functor (function symbol of arity $n > 0$ stands for a function from n entities to one entity in the domain
  - each predicate of arity n stands for a particular relationship between n entities in the domain of discourse

Views of predicates

- predicate with n arguments can be viewed in different ways:

    - function from all possible combinations of n terms to a truth value
    - set of tuples of n terms. Every tuple in the set is implicitly mapped to true, while every other tuple is mapped to false

- predicate definition is then, for each view
    - define the mapping
    - define the set of tuples

Meaning of clauses

```
grandparent(A, C) :- parent(A,B), parent(B,C).
```

means "for all terms that A and C may stand for: A is a grandparent of C if there is a term B such that A is a parent of B, and B is a parent of C"

$$\forall A, C(grandparent(A, C) \leftarrow \exists B(parent(A, B) \wedge parent(B, C))).$$

- variables in the head are universally quantified over the entire clause
- variables appearing only in the body are existentially quantified over the body

Meaning of predicate definitions

- predicates are define by a finite number of clauses, each of which is in the form of an implication
- e.g. `parent(queen_elizabeth, prince_charles)` represents

$$\forall A, B(parent(A, B) \leftarrow (A = \text{queen\_elizabeth} \wedge B = \text{prince\_charles}))$$

- the meaning of the predicate is a disjunction of the bodies of all the clauses:

$$\forall A, B : parent(A, B) \leftarrow (A = \text{queen\_elizabeth} \wedge B = \text{prince\_charles}) \vee (A = \text{prince\_philip} \wedge B = \text{prince\_cha}$$

Closed world assumption

- to implement closed world assumption, make the implication biimplication

$$\forall A, B : parent(A, B) \iff (A = \text{queen\_elizabeth} \wedge B = \text{prince\_charles}) \vee (A = \text{prince\_philip} \wedge B = \text{prince\_c}$$

- A is not a parent of B unless they are one of the listed cases

- adding reverse implication produces the **Clark completion** of the program

Semantics

- logic program P consists of a set of predicate definitions
- **semantics/meaning of P**: set of **logical consequences** as ground atomic formulae
- a ground atomic formula a is a logical consequence of a program P if P makes it true
- a negated ground atomic formula $\neg a$ (`\+a`) is a logcical consequence of P if a is not a logical conequence of P

Finding semantics

- you can determine the semantics of a logic program by working backwards: instead of reasoning from a query to find a satisfying substitution, you reason from the program to find what ground queries succeed.
- immediate consequence operator T_P takes a set of ground unit clauses C and produces the set of ground unit clauses implied by C together with the program P
- always includes all ground instances of all unit clauses in P
- for each clause `H :- G1, ... Ga` in P

e.g.

```
% P
q(X,Z) :- p(X,Y), p(Y,Z).
% C
p(a,b).
p(b,c).
p(c,d).
```

Then:

$$T_P(C) = \{q(a,c).q(b,d).\}$$

- the semantics of P is always T_P applied infinitely many times to the empty set: $T_P(T_P(...()...))$

Procedural interpretation

```
grandparent(A, C) :- parent(A,B), parent(B,C).
```

Logical interpretation:

$$\forall A, C(grandparent(A,C) \leftarrow \exists B(parent(A,B) \wedge parent(B,C))).$$

Procedural interpretation: to show that A is a grandparent of C, it suffices to show A is a parent of B and B is a parent of C.

Selective Linear Definite Resolution

- consequences of a logic program are determined through **resolution**
- **SLD resolution** is an efficient version of resolution
- e.g. to determine if Queen Elizabeth is Prince Harry's grandparent:

```
?- grandparent(queen_elizabeth, prince_harry).
```

With program:

```
granpdarent(X,Z) :_ parent(X,Y), parent(Y,Z).
```

- Unify query goal (`grandparent(queen_elizabeth, prince_harry)` with clause head `grandparent(X,Z)`
- apply resulting substitution to the clause, to yield the **resolvent**
- the goal is identical to the resolvent head, so we can replace it with the resolvent body:

```
?- parent(queen_elizabeth, Y), parent(Y, prince_harry).
```

- now pick one of these goals to resolve: say you choose `parent(Y, prince_harry)`
- only two clauses can resolve with it:

```
parent(prince_charles, prince_harry).
parent(princess_diana, prince_harry).
```

- choose the second clause. After resolution, we are left with the query:

```
?- parent(queen_elizabeth, princess_diana).
```

- no clause unifies with this query: resolution fails. This can sometimes take many steps.
- now **backtrack** and try the first matching clause

```
?- parent(queen_elizabeth, prince_charles).
```

- there is a matching program clause, so there is nothing more to prove. The query succeeds.

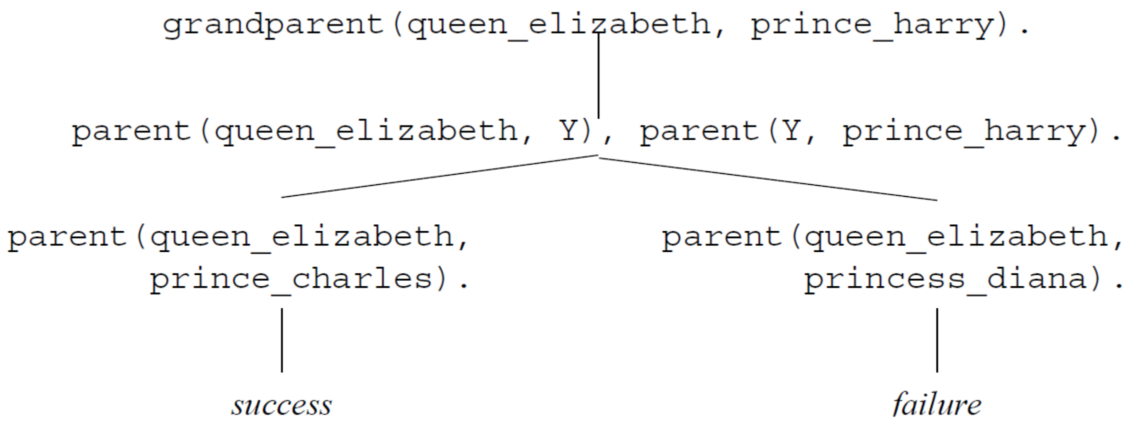
grandparent(queen_elizabeth, prince_harry).
parent(queen_elizabeth, Y), parent(Y, prince_harry).
parent(queen_elizabeth, prince_charles).
parent(queen_elizabeth, princess_diana).
success
failure

Figure 1: SLD Tree

Order of Execution

- order of goal resolution and trial of different clauses:
  - doesn't matter for correctness
  - does matter for efficiency
- Prolog always selects the first goal to resolve, and always selects the first matching clause to pursue first
  - allows programmer to control execution

Backtracking

- when there are multiple clauses matching a goal, Prolog leaves a **choicepoint** so that it can return to that state and try the next matching clause
- when a goal fail, Prolog **backtracks** to the most recent choicepoint
  - removes all variable bindings since the choicepoint
  - Prolog begins resolution with next matching clause
  - once all matching clauses are exhausted, the choicepoint is removed
  - subsequent failures then backtrack to the next choicepoint

Indexing

- **indexing** can improve efficiency

- Prolog systems automatically create an index for predicates with multiple clauses, where the heads have distinct constants/functors
- for a call with the first argument called, Prolog immediately jumps to the first matching clause
- SWI Prolog constructs indices for multiple arguments, meaning more queries benefit from indexing

Debugging

Prolog Debugger

- `trace` turns on the debugger
- `nodebug` turns off the debugger
- Byrd box model: goal execution is a box with ports for entry/exit

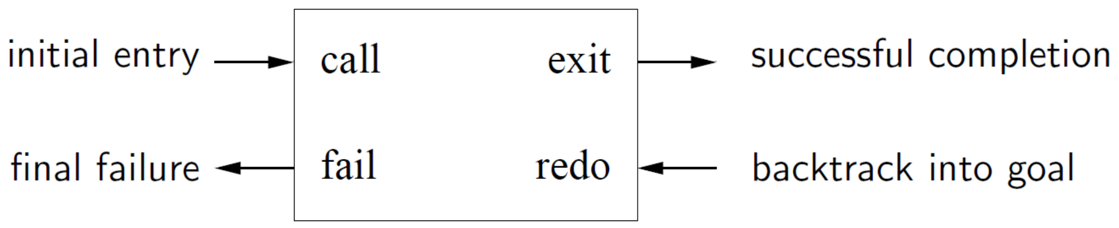

Figure 2: Prolog Debugger

Infinite Backtracking Loop

Initial version of reverse

```prolog
% rev1(X,Y)
% rev1/2 holds when Y is the reverse of the list X
rev1([], []).
rev1([A|BC], CBA) :-
    rev1(BC, CB),
    append(CB, [A], CBA).
```

- Doesn't work if the first argument is free, e.g. `rev1(X, [a]).`
- Prolog enters an infinite backtracking loop:
    - `rev1(BC,CB)` has an infinite backtracking sequence of solutions `{BC ->[Z], CB->[Z]}, {BC ->[Y,Z], CB->[Z,Y]}`
    - `append([Z], [A], [a]` fails

    - `append([Y,Z], [A], [a]` fails
    - ...

- you could prevent this by swapping the body goals around, but then it won't work with the second argument free
- solution: ensure that `rev1`'s first argument is always bound to a list when called

    - length of a list must always be the same as that of its reverse
    - when `same_length/2` succeeds, both arguments are bound to lists of the same fixed length

```prolog
% rev3(X,Y)
% rev3/2 holds when Y is the reverse of the list X
rev3(ABC, CBA) :-
    same_length(ABC, CBA),
    rev1(ABC, CBA).

same_length([], []).
same_length([_|Xs], [_|Ys]) :-
    same_length(Xs, Ys).
```

Managing nondeterminism

- when clauses succeed, but there are later clauses that may succeed, Prolog leaves a choicepoint so that it can later backtrack and try the later clause
- when efficiency matters: ensure recursive predicates don't leave choicepoints when they should be deterministic (i.e. leave no choice points) when there are no other solutions
- if choicepoints remain, it disables tail recursion optimisation

If-then-else `( -> ; ).`

- can be used to avoid choicepoints
- `->` is treated like conjunction, but any alternative solutions of the condition, and any alternatives of the else block will be forgotten.

- if the condition goal fails, the else goal is tried
- deterministic whenever the then/else blocks are deterministic
- indexing is preferable to ITE: avoid where possible

    - ITE often prevents code working in multiple modes

Tail Recursion

- **tail recursive:** the only recursive call is the last code executed before returning to the caller

Tail Recursion Optimisation (TRO)

- Prolog performs **tail recursion optimisation**, making recursive predicates behave as if they were loops
- more often applicable in Prolog than other languages

The Stack

- **stack frame:** stores local variables and where to return to when finished
- when a calls b, it creates a fresh stack frame for b, preserving a's frame
- similarly when b calls c
- if all b does after calling c is return to a, there is no need to preserve b's local variables

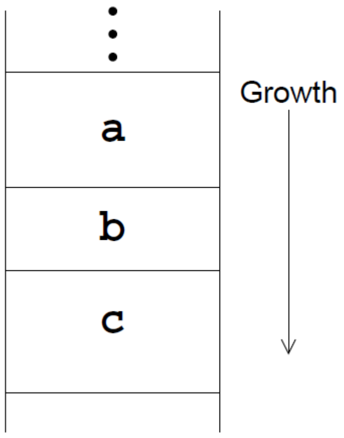

Figure 3: Stack frames

- **last call optimisation:** saves significant stack space
  - Prolog can release b's frame before calling c
  - then when c finishes, it directly returns to a

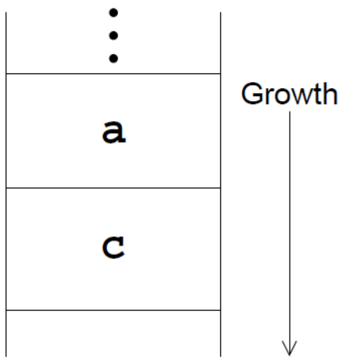

Figure 4: Last call optimisation

- TRO is a special case of last call optimisation, where the last call is recursive
- particularly beneficial, as recursion replaces looping
- without TRO: a new stack frame would be needed for each iteration, which would quickly exhaust the stack
- with TRO: tail recursive predicates execute in **constant** stack space, just like a loop

Choicepoints

- if b leaves a choicepoint, it sits on the stack above b's frame
- this freezes it, and all earlier frames, meaning they can't be reclaimed
- this is necessary: when Prolog backtracks to the choicepoint, b's arguments must be ready to try the next matching clause for b

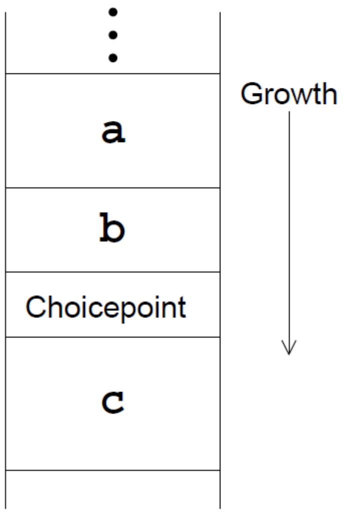

Figure 5: TRO and choicepoints

Accumulator

- make code tail recursive through an **accumulating parameter/accumulator**, an extra parameter to the predicate that holds a partially computed result
- base case: (usually) partially computed result is the actual result
- recursive clause: compute more of the partially computed result, and pass this to the recursive goal
- helpful approach: consider how you would implement it using a `while` loop, then implement it in Prolog

Accumulating Lists

- accumulators can make an order difference to efficiency
  - e.g. replacing `append/3` (linear time) with list construction (constant time)
- e.g. `rev1` defined earlier is $O(n^2)$:
  - for the n-th element from the end of we append a list of length $n-1$ to a singleton list
  - doing this n times gives ~ $\frac{n(n-1)}{2}$

Tail Recursive `rev/2`

```prolog
% rev(BCD, A, DCBA)
% DCBA is BCD reversed, with A appended
rev([], A, A).
rev([B|CD], A, DCBA) :-
  rev(CD, [B|A], DCBA).
```

- at each step, an element is removed from the head of the input list, and added to the head of the accumulator
- the cost of each step is constant, so overall cost is linear in length of the list
- accumulator here works like a stack, last element is the first element added to it

Difference Pairs

- the tail recursive `rev` is a common pattern in Prolog
- a predicate that generates a list takes an extra argument specifying what should come after the list, avoiding the need to append to the list
- if you don't know what will come after the list at the time you call the predicate, you can pass an unbound variable
- then bind the variable when you do know what should come after
- many predicates intended to produce a list have 2 arguments
  - 1st argument: list produced
  - 2nd argument: what comes after
- **difference pair:** predicate that generates the difference between the first and second list

```prolog
% inorder tree flatten
flatten(empty, List, List).
flatten(node(L,E,R), List, List0) :-
    flatten(L, List, List1),
    List1 = [E|List2],
    flatten(R, List2, List0).
```

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.
```

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.

```
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
`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)

Function	Description
`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

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 ,

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