Programs
Horn clauses and unification, with backtracking
conclusion :- condition, condition, condition.

Grammar
Term ::= Constant | CompoundTerm | Variable
Constant ::= Atom | Number
CompoundTerm ::=Term(Term+)
Variables start with capital letters
Atoms start with small letters

Operators
Define using op(precedence, associativity, name).
Built in: , (and), ; (or), is (arithmetic expression , RHS must be ground), =:=, =\= ([not] arithmetic equal), <=, >= (arithmetic smaller / larger), =, \= ([not] unifiable), ==, \==, ([not] identical)
Note: "is" cannot solve equations

Lists
Cons = .(head, tail), Empty= []
Alternate syntax: [a, b, c | Rest]

Cut
! : controls backtracking. When the cut is "crossed" backtracking over it will cause the clause to fail
"fail" always fails, so you can use !, fail to fail fast
Green cuts prune the search tree (as in "member", later)
Red cuts omit explicit conditions: program becomes special cases and a default rule

Negation
not(G) :- call(G), !, fail.
Negation is failure in Prolog, do not confuse this with logical negation! For example:
innocent(a).
?- innocent(b). = <No>
Solve by storing negative information, adding a clause saying that if it occurs once it cannot occur anywhere else (e.g. a battle only occurs in 1815, then don't both adding "battle didn't occur in 1816" etc) or make the *closed world assumption* (everything that exists is stated in the program) and implement negation by failure.

Extras
_ represents unused variable
Declare rule with "dynamic ruleName\arity" e.g. "dynamic awayGoal\2"
Use assert/retract to introduce clauses into Prolog at runtime

Basic Examples

```
length([], 0).
length([H|T], N) :- length(T, NT), N is NT + 1.
length2(L, N) :- acc(L, 0, N).
acc([], A, A).
acc([H|T], A, N) :- A1 is A + 1, acc(T, A1, N). (tail recursive)
max([], A, A).
max([H|T], A, M) :- H > A, max(T, H, M).
max([H|T], A, M) :- H =< A, max(T, A, M). (initialise the accumulator with the list head)
reverse([], []).
reverse([H|T], O) :- reverse(T, Y), append(Y, [H], O).
reverse2([], O, O).
reverse2([H|T], A, O) :- reverse2(T, [H|A], O). (tail rec.)
```

Partial Maps

```
partial([], []).
partial([X|T], [X|L]) :- include(X), partial(T, L).
partial([X|T], L) :- partial(T, L).
partial2([], [], []).
partial2([H|T], [H|A], B) :- inA(H), partial2(T, A, B).
partial2([H|T], A, [H|B]) :- partial2(T, A, B).
```

Graphs
Represent edges as a(u,v) or u-v

Binary Search Tree

```
mt. % empty node
n(L,Value,R). % node functor

insert(Item, n(L, Item, R), n(L, Item, R)).
insert(Item, mt, n(L, Item, R)).
insert(Item, n(L, T, R), n(NL, T, R)) :- Item < T, insert(Item, L, NL).
insert(Item, n(L, T, R), n(L, T, NR)) :- Item > T, insert(Item, R, NR).
```

Using Cut

```
member(X, [X|_]) :- !.
member(X, [_|L]) :- member(X, L).
```

Warning:

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

max(10, 0, 0) = Yes
max(10, 0, X) = 10, <No>

	Use: max(X, Y, X) :- X >= Y, !. max(X, Y, Y) :- X < Y, !.
Difference Lists	X-X is an empty list Change list X to a normal list by unifying it with Y-[] append(A-B, B-C, A-C). (constant time append: execute as append([a, b|Y]-Y, [c, d|Z]-Z, X-XS).) reverse([]-[], []-[]). reverse([H1|T]-End1, Result) :- reverse(T-End1, SubResult), append(SubResult, [H1|X]-X, Result). Better, not using append: reverse([]-[], X-X). reverse([H|T]-End1, Start2-End2) :- reverse(T-End1, Start2-[Head|End2]).