Writing Prolog Programs

Hao Su

July 5, 2025

Cliches

Elegance is not optional.
- The Craft of Prolog

Cliches

> *Elegance is not optional.*
> *- The Craft of Prolog*

If your Prolog code is ugly, the chances are that you either don't understand your problem or you don't understand your programming language, and in neither case does your code stand much chance of being efficient.

Cliches

Elegance is not optional.
- The Craft of Prolog

If your Prolog code is ugly, the chances are that you either don't understand your problem or you don't understand your programming language, and in neither case does your code stand much chance of being efficient.

Keep your code clear and straightforward for ease of maintenance.

Table of Contents

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*.
Our task is to *describe* the situation.

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*. Our task is to *describe* the situation.

It is so-called *declarative programming*: We state clearly what we know about the task, whereas in imperative programming we state what must be done.

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*. Our task is to *describe* the situation.

It is so-called *declarative programming*: We state clearly what we know about the task, whereas in imperative programming we state what must be done.

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*. Our task is to *describe* the situation.

It is so-called *declarative programming*: We state clearly what we know about the task, whereas in imperative programming we state what must be done.

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

We focus on the *logic*: What is true and what follows from what.

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*. Our task is to *describe* the situation.

It is so-called *declarative programming*: We state clearly what we know about the task, whereas in imperative programming we state what must be done.

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

We focus on the *logic*: What is true and what follows from what.

The prolog engine derives *logical consequences* from the code, and we are not concerned with how it does so.

A Broad Discussion

Prolog code states *what holds*. It is an *executable specification*. Our task is to *describe* the situation.

It is so-called *declarative programming*: We state clearly what we know about the task, whereas in imperative programming we state what must be done.

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

We focus on the *logic*: What is true and what follows from what.

The prolog engine derives *logical consequences* from the code, and we are not concerned with how it does so.

A major attraction of Prolog is that we can easily switch strategies/implementations, i.e., Prolog performs controlled deduction.

Case Study: Concatenation

Goal: Define the predicate `list_list_together/3`.

Case Study: Concatenation

Goal: Define the predicate `list_list_together/3`.

- We describe a relation over *lists*.
- Two structural cases:
  - The empty list: `[]`
  - A non-empty list: `[L|Ls]`

Case Study: Concatenation

Goal: Define the predicate `list_list_together/3`.

- We describe a relation over *lists*.
- Two structural cases:
  - The empty list: `[]`
  - A non-empty list: `[L|Ls]`

Initial structure:

```
list_list_together([], Bs, Cs) :-
   ...
list_list_together([L|Ls], Bs, Cs) :-
   ...
```

Case Study: Concatenation

Goal: Define the predicate `list_list_together/3`.

- We describe a relation over *lists*.
- Two structural cases:
  - The empty list: `[]`
  - A non-empty list: `[L|Ls]`

Initial structure:

```
list_list_together([], Bs, Cs) :-
   ...
list_list_together([L|Ls], Bs, Cs) :-
   ...
```

Ask: Under what conditions do these clauses hold?

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Cs) :-
    ...
list_list_together([L|Ls], Bs, Cs) :-
    ...
```

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Cs) :-
    ...
list_list_together([L|Ls], Bs, Cs) :-
    ...
```

Step 1: Base case: empty list.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Cs) :-
    Bs = Cs.
list_list_together([L|Ls], Bs, Cs) :-
    ...
```

Step 1: Base case: empty list.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Bs) :-

list_list_together([L|Ls], Bs, Cs) :-
   ...
```

Step 1: Base case: empty list.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Bs) :-

list_list_together([L|Ls], Bs, Cs) :-
   ...
```

Step 1: Base case: empty list.

Step 2: Recursive case.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Bs) :-

list_list_together([L|Ls], Bs, Cs) :-
  Cs = [L|Rest],
  list_list_together(Ls, Bs, Rest).
```

Step 1: Base case: empty list.

Step 2: Recursive case.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Bs) :-

list_list_together([L|Ls], Bs, Cs) :-
  Cs = [L|Rest],
  list_list_together(Ls, Bs, Rest).
```

Step 1: Base case: empty list.

Step 2: Recursive case.

Step 3: Cleaning.

Case Study: Concatenation

Ask: Under what conditions do these clauses hold?

```
list_list_together([], Bs, Bs) :-

list_list_together([L|Ls], Bs, [L|Rest],) :-
  list_list_together(Ls, Bs, Rest).
```

Step 1: Base case: empty list.

Step 2: Recursive case.

Step 3: Cleaning.

Some other cliches

Ask NOT what should Prolog *do* in some case.

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.
Longer answer: think of relations between different *states* of something, and the state conditions that make this hold. So, what are the relations between a list and a list some certain element is removed?

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.
Longer answer: think of relations between different *states* of something, and the state conditions that make this hold. So, what are the relations between a list and a list some certain element is removed?

Another Example: How do we test if two nodes in an undirected graph are reachable from each other?

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.
Longer answer: think of relations between different *states* of something, and the state conditions that make this hold. So, what are the relations between a list and a list some certain element is removed?

Another Example: How do we test if two nodes in an undirected graph are reachable from each other? BFS?

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.
Longer answer: think of relations between different *states* of something, and the state conditions that make this hold. So, what are the relations between a list and a list some certain element is removed?

Another Example: How do we test if two nodes in an undirected graph are reachable from each other? BFS?

```
reachable(S, S).
reachable(S0, S) :- edge(S0, S1), reachable(S1, S).
```

Some other cliches

Ask NOT what should Prolog *do* in some case. Instead ask what conditions make the relation *hold*.

Example. How do we remove some element in Prolog?

Short answer, we don't.
Longer answer: think of relations between different *states* of something, and the state conditions that make this hold. So, what are the relations between a list and a list some certain element is removed?

Another Example: How do we test if two nodes in an undirected graph are reachable from each other? BFS?

```
reachable(S, S).
reachable(S0, S) :- edge(S0, S1), reachable(S1, S).
```

Implication: a clear understanding of your problem, often mathematical, is all you need.

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`
2. resource leaks

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`
2. resource leaks `setup_call_cleanup/3`

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`
2. resource leaks `setup_call_cleanup/3`
3. efficient `dif/2`

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`
2. resource leaks `setup_call_cleanup/3`
3. efficient `dif/2` `if_/3`

Prolog is Growing

There are Prolog programs that cannot *yet* be written satisfactorily.

Reason: Not all useful language constructs are yet found.

Past chanllenges resolved by new language constructs:

1. efficient parsing `DCGs`
2. resource leaks `setup_call_cleanup/3`
3. efficient `dif/2` `if_/3`
4. ...

Caveats

- Ideally, a predicate can be used in all directions, so we choose *nouns* in lowercases to denote them.

Caveats

- Ideally, a predicate can be used in all directions, so we choose *nouns* in lowercases to denote them.
- `using_underscores_makes_longer_names_easy_to_read`

Caveats

- Ideally, a predicate can be used in all directions, so we choose *nouns* in lowercases to denote them.
- `using_underscores_makes_longer_names_easy_to_read`
- `list_length/2` versus `length/2`

Caveats

- Ideally, a predicate can be used in all directions, so we choose *nouns* in lowercases to denote them.
- `using_underscores_makes_longer_names_easy_to_read`
- `list_length/2` versus `length/2`
- A variable starts with an uppercase letter or with an underscore.

Caveats

- Ideally, a predicate can be used in all directions, so we choose *nouns* in lowercases to denote them.
- `using_underscores_makes_longer_names_easy_to_read`
- `list_length/2` versus `length/2`
- A variable starts with an uppercase letter or with an underscore.
- State transitions should be named like

$$\texttt{State0} \to \texttt{State1} \to \texttt{State2} \to \cdots \to \texttt{State}.$$

Table of Contents

Termination

There are two kinds of termination:

Termination

There are two kinds of termination:

- Existential Termination

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.
  - Prolog is relational programming language, so *existential* termination does not fully characterize the procedural behavior of a Prolog query.

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.
  - Prolog is relational programming language, so *existential* termination does not fully characterize the procedural behavior of a Prolog query.
- Universal Termination

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.
  - Prolog is relational programming language, so *existential* termination does not fully characterize the procedural behavior of a Prolog query.
- Universal Termination
  - A Prolog query `Q` terminates *universally* iff the query `?- Q, false.` terminates existentially.

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.
  - Prolog is relational programming language, so *existential* termination does not fully characterize the procedural behavior of a Prolog query.
- Universal Termination
  - A Prolog query `Q` terminates *universally* iff the query `?- Q, false.` terminates existentially.
    How do understand this?

Termination

There are two kinds of termination:

- Existential Termination
  - A Prolog query `Q` terminates *existentially* iff we receive an answer when posting `Q`.
  - Prolog is relational programming language, so *existential* termination does not fully characterize the procedural behavior of a Prolog query.
- Universal Termination
  - A Prolog query `Q` terminates *universally* iff the query `?- Q, false.` terminates existentially.
    How do understand this?

Both are *undecidable*.

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

```
adjacent(a, b).
adjacent(e, f).
adjacent(X, Y) :- adjacent(Y, X).

?- adjacent(X, Y), false.
```

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

Debug: Faliure Slicing

```
adjacent(a, b).
adjacent(e, f).
adjacent(X, Y) :- adjacent(Y, X).

?- adjacent(X, Y), false.
```

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

Debug: Faliure Slicing

```
adjacent(a, b) :- false.
adjacent(e, f) :- false.
adjacent(X, Y) :- adjacent(Y, X).

?- adjacent(X, Y), false.
```

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

Debug: Faliure Slicing

```
adjacent(a, b) :- false.
adjacent(e, f) :- false.
adjacent(X, Y) :- adjacent(Y, X).

?- adjacent(X, Y), false.
```

Still Nontermination!

Programs may Not Terminate

- A program may never *terminate* when, for example, it is written to *generate* objects from an infinite set.
- Programs may search for objects whose very existence is not a priori known.
- Or... we might have wrongly composed our programs...

Debug: Faliure Slicing

```
adjacent(a, b) :- false.
adjacent(e, f) :- false.
adjacent(X, Y) :- adjacent(Y, X).

?- adjacent(X, Y), false.
```

Still Nontermination!

How to resolve this then?

Table of Contents

Sorting

The standard order of terms:

variables $\prec$ numbers $\prec$ atoms $\prec$ compound terms

Sorting

The standard order of terms:
$\text{variables} \prec \text{numbers} \prec \text{atoms} \prec \text{compound terms}$

Use `@</2`, `@=</2`, `compare/3`, and other predicates to compare terms according to the standard order.

Sorting

The standard order of terms:
variables $\prec$ numbers $\prec$ atoms $\prec$ compound terms

Use `@</2`, `@=</2`, `compare/3`, and other predicates to compare terms according to the standard order. They are not true relations though. For example:

```
?- a @< b.
  true.

?- a @< X.
  false.
```

Sorting

The standard order of terms:

$\text{variables} \prec \text{numbers} \prec \text{atoms} \prec \text{compound terms}$

Use `@</2`, `@=</2`, `compare/3`, and other predicates to compare terms according to the standard order. They are not true relations though. For example:

```
?- a @< b.
  true.

?- a @< X.
  false.
```

`sort(Ls0, Ls)`: Both are lists. True iff `Ls` holds the elements of the list `Ls0` sorted according to the standard order of terms and contains *no duplicates*.

Sorting

The standard order of terms:
variables $\prec$ numbers $\prec$ atoms $\prec$ compound terms

Use `@</2`, `@=</2`, `compare/3`, and other predicates to compare terms according to the standard order. They are not true relations though. For example:

```
?- a @< b.
   true.

?- a @< X.
   false.
```

`sort(Ls0, Ls)`: Both are lists. True iff `Ls` holds the elements of the list `Ls0` sorted according to the standard order of terms and contains *no duplicates*.

`keysort(Pairs0, Pairs)`: Both are key-valuse pairs. True iff `Pairs0` is sorted by `Key`. Duplicates are *retained*. *Stable*.

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])
```

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])
```

```
list_length([], 0).
list_length([_|Ls], Length) :-
        Length #= Length0 + 1,
        list_length(Ls, Length0).
```

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])
```

```
list_length([], 0).
list_length([_|Ls], Length) :-
        Length #= Length0 + 1,
        list_length(Ls, Length0).

list_pair(Ls, L-Ls) :-
        list_length(Ls, L).
```

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])
```

```
list_length([], 0).
list_length([_|Ls], Length) :-
        Length #= Length0 + 1,
        list_length(Ls, Length0).

list_pair(Ls, L-Ls) :-
        list_length(Ls, L).

?- lists(Lists),
   maplist(list_pair, Lists, Pairs0),
   keysort(Pairs0, Pairs).
   Lists = ["abcd","abc","abcde","a","ab"],
   Pairs0 = [4-"abcd",3-"abc",5-"abcde",1-"a",2-"ab"],
```

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])

list_length([], 0).
list_length([_|Ls], Length) :-
        Length #= Length0 + 1,
        list_length(Ls, Length0).

list_pair(Ls, L-Ls) :-
        list_length(Ls, L).

?- lists(Lists),
   maplist(list_pair, Lists, Pairs0),
   keysort(Pairs0, Pairs).
   Lists = ["abcd","abc","abcde","a","ab"],
   Pairs0 = [4-"abcd",3-"abc",5-"abcde",1-"a",2-"ab"],

Pairs = [1-"a",2-"ab",3-"abc",4-"abcd",5-"abcde"].
```

Example: Sorting lists by their Lengths

```
lists(["abcd", "abc", "abcde", "a", "ab"])
```

```
list_length([], 0).
list_length([_|Ls], Length) :-
        Length #= Length0 + 1,
        list_length(Ls, Length0).

list_pair(Ls, L-Ls) :-
        list_length(Ls, L).

?- lists(Lists),
   maplist(list_pair, Lists, Pairs0),
   keysort(Pairs0, Pairs).
   Lists = ["abcd","abc","abcde","a","ab"],
   Pairs0 = [4-"abcd",3-"abc",5-"abcde",1-"a",2-"ab"],
```

Pairs = [1-"a",2-"ab",3-"abc",4-"abcd",5-"abcde"].

Write `Pairs = [_-Ls|_]` to obtain a list with minimum length.
etc.

Quicksorts

```
quicksort([], []).  % Base case: empty list is already
     sorted
quicksort([Pivot|Rest], Sorted) :-
    partition(Rest, Pivot, Less, Greater),
    quicksort(Less, SortedLess),
    quicksort(Greater, SortedGreater),
    append(SortedLess, [Pivot|SortedGreater], Sorted).

partition([], _, [], []).
partition([X|Xs], Pivot, [X|Ls], Gs) :-
    X =< Pivot,
    partition(Xs, Pivot, Ls, Gs).
partition([X|Xs], Pivot, Ls, [X|Gs]) :-
    X > Pivot,
    partition(Xs, Pivot, Ls, Gs).
```

Quicksorts

```
quicksort([], []).  % Base case: empty list is already
     sorted
quicksort([Pivot|Rest], Sorted) :-
    partition(Rest, Pivot, Less, Greater),
    quicksort(Less, SortedLess),
    quicksort(Greater, SortedGreater),
    append(SortedLess, [Pivot|SortedGreater], Sorted).

partition([], _, [], []).
partition([X|Xs], Pivot, [X|Ls], Gs) :-
    X =< Pivot,
    partition(Xs, Pivot, Ls, Gs).
partition([X|Xs], Pivot, Ls, [X|Gs]) :-
    X > Pivot,
    partition(Xs, Pivot, Ls, Gs).

quicksort :: (Ord a) => [a] -> [a]
quicksort []     = []
quicksort (p:xs) =
  let smallerSorted = quicksort [x | x <- xs, x <= p]
      biggerSorted  = quicksort [x | x <- xs, x > p]
  in  smallerSorted ++ [p] ++ biggerSorted
```

Searching

- Prolog is good at searching.

Searching

- Prolog is good at searching.
  - Prolog's execution strategy is already a form of *search*: DFS with chronological backtracking.

Searching

- Prolog is good at searching.
  - Prolog's execution strategy is already a form of *search*: DFS with chronological backtracking.
  - Other search strategies can be readily implemented on top of the built-in strategy.

Searching

- Prolog is good at searching.
  - Prolog's execution strategy is already a form of *search*: DFS with chronological backtracking.
  - Other search strategies can be readily implemented on top of the built-in strategy.

An example: Search for the transitive closure of a node in a graph.

Searching

- Prolog is good at searching.
  - Prolog's execution strategy is already a form of *search*: DFS with chronological backtracking.
  - Other search strategies can be readily implemented on top of the built-in strategy.

An example: Search for the transitive closure of a node in a graph.

```
k_n(N, Adjs) :-
        list_length(Nodes, N),
        Nodes ins 1..N,
        all_distinct(Nodes),
        once(label(Nodes)),
        maplist(adjs(Nodes), Nodes, Adjs).

adjs(Nodes, Node, Node-As) :-
        tfilter(dif(Node), Nodes, As).
```

Searching

- Prolog is good at searching.
  - Prolog's execution strategy is already a form of *search*: DFS with chronological backtracking.
  - Other search strategies can be readily implemented on top of the built-in strategy.

An example: Search for the transitive closure of a node in a graph.

```
k_n(N, Adjs) :-
        list_length(Nodes, N),
        Nodes ins 1..N,
        all_distinct(Nodes),
        once(label(Nodes)),
        maplist(adjs(Nodes), Nodes, Adjs).

adjs(Nodes, Node, Node-As) :-
        tfilter(dif(Node), Nodes, As).

?- k_n(3, Adjs).
   Adjs = [1-[2,3],2-[1,3],3-[1,2]]
;  ... .
```

Example: Inefficient Transitive Closure

```
reachable(_, _, From, From).
reachable(Adjs, Visited, From, To) :-
        maplist(dif(Next), Visited),
        member(From-As, Adjs),
        member(Next, As),
        reachable(Adjs, [From|Visited], Next, To).
```

Example: Inefficient Transitive Closure

```
reachable(_, _, From, From).
reachable(Adjs, Visited, From, To) :-
        maplist(dif(Next), Visited),
        member(From-As, Adjs),
        member(Next, As),
        reachable(Adjs, [From|Visited], Next, To).

?- k_n(3, Adjs),
   setof(To, reachable(Adjs, [], 1, To), Tos).
   Adjs = [1-[2,3],2-[1,3],3-[1,2]], Tos = [1,2,3]
;  false.
```

Example: Inefficient Transitive Closure

```
reachable(_, _, From, From).
reachable(Adjs, Visited, From, To) :-
        maplist(dif(Next), Visited),
        member(From-As, Adjs),
        member(Next, As),
        reachable(Adjs, [From|Visited], Next, To).

?- k_n(3, Adjs),
   setof(To, reachable(Adjs, [], 1, To), Tos).
   Adjs = [1-[2,3],2-[1,3],3-[1,2]], Tos = [1,2,3]
;  false.

?- list_length(_, N), portray_clause(N),
   k_n(N, Adjs),
   time(setof(To, reachable(Adjs, [], 1, To), Tos)),
   false.
...
7.
   % CPU time: 1.454s
8.
   % CPU time: 13.628s
```

Example: Warshall's algorithm

```
warshall(Adjs, Nodes0, Nodes) :-
        phrase(reachables(Nodes0, Adjs), Nodes1,
            Nodes0),
        sort(Nodes1, Nodes2),
        if_(Nodes2 = Nodes0,
            Nodes = Nodes2,
            warshall(Adjs, Nodes2, Nodes)).

reachables([], _) --> [].
reachables([Node|Nodes], Adjs) -->
        { member(Node-Rs, Adjs) },
        Rs,
        reachables(Nodes, Adjs).
```

Example: Warshall's algorithm

```
warshall(Adjs, Nodes0, Nodes) :-
        phrase(reachables(Nodes0, Adjs), Nodes1,
            Nodes0),
        sort(Nodes1, Nodes2),
        if_(Nodes2 = Nodes0,
            Nodes = Nodes2,
            warshall(Adjs, Nodes2, Nodes)).

reachables([], _) --> [].
reachables([Node|Nodes], Adjs) -->
        { member(Node-Rs, Adjs) },
        Rs,
        reachables(Nodes, Adjs).

?- k_n(9, Adjs),
   time(warshall(Adjs, [1], Tos)).
   % CPU time: 0.000s
   ...,
   Tos = [1,2,3,4,5,6,7,8,9]
;  ... .
```

Table of Contents

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Denote a *knave* with a *0*, a knight a *1*, and one Boolean *variable* each inhabitant.

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Denote a *knave* with a *0*, a knight a *1*, and one Boolean *variable* each inhabitant.

Eg1: You meet 2 inhabitants, A and B. A says: "Either I am a knave or B is a knight."

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Denote a *knave* with a *0*, a knight a *1*, and one Boolean *variable* each inhabitant.

Eg1: You meet 2 inhabitants, A and B. A says: "Either I am a knave or B is a knight."

```
?- sat(A =:= ~A+B).
   A = 1, B = 1.
```

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Denote a *knave* with a *0*, a knight a *1*, and one Boolean *variable* each inhabitant.

Eg1: You meet 2 inhabitants, A and B. A says: "Either I am a knave or B is a knight."

```
?- sat(A =:= ~A+B).
   A = 1, B = 1.
```

Eg2: A says: "B is a knave." B says: "A and C are of the same kind." What is C?

Knights and Knaves

Premise: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You are to tell them apart.

Denote a *knave* with a *0*, a knight a *1*, and one Boolean *variable* each inhabitant.

Eg1: You meet 2 inhabitants, A and B. A says: "Either I am a knave or B is a knight."

```
?- sat(A =:= ~A+B).
   A = 1, B = 1.
```

Eg2: A says: "B is a knave." B says: "A and C are of the same kind." What is C?

```
?- sat(A =:= ~B), sat(B =:= (A=:=C)).
   C = 0, clpb:sat(A=\=B).
```

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

None of the unnoticed things, met with at sea, are mermaids.

Things entered in the log, as met with at sea, are sure to be worth remembering.
I have never met with anything worth remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

None of the unNoticed things, met with at sea, are Mermaids.

Things entered in the log, as met with at sea, are sure to be worth remembering.
I have never met with anything worth remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \to N$, i.e. `sat(M =< N)`.

Things entered in the log, as met with at sea, are sure to be worth remembering.
I have never met with anything worth remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \to N$, i.e. `sat(M =< N)`.

Things entered in the Log, as met with at sea, are sure to be worth Remembering.
I have never met with anything worth remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \to N$, i.e. `sat(M =< N)`.

$L \to R$, i.e. `sat(L =< R)`.

I have never met with anything worth remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \rightarrow N$, i.e. `sat(M =< N).`

$L \rightarrow R$, i.e. `sat(L =< R).`

I have never met with anything worth Remembering, when on a voyage.
Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \rightarrow N$, i.e. `sat(M =< N)`.

$L \rightarrow R$, i.e. `sat(L =< R)`.

$I \rightarrow \neg R$, i.e. `sat(I =< ~R)`.

Things met with at sea, that are noticed, are sure to be recorded in the log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \to N$, i.e. `sat(M =< N)`.

$L \to R$, i.e. `sat(L =< R)`.

$I \to \neg R$, i.e. `sat(I =< ~R)`.

Things met with at sea, that are Noticed, are sure to be recorded in the Log.

Lewis Carroll

In many of his puzzles, your job is to string together all given statements so that they form a chain of implications, typically arriving at a result that is surprising or amusing. For example:

$M \to N$, i.e. `sat(M =< N).`

$L \to R$, i.e. `sat(L =< R).`

$I \to \neg R$, i.e. `sat(I =< ~R).`

$N \to L$, i.e. `sat(N =< L).`

Lewis Carroll

```
sea([N,M,L,R,I]) :-
        sat(M =< N),    % statement 1
        sat(L =< R),    % statement 2
        sat(I =< ~R),   % statement 3
        sat(N =< L).    % statement 4
```

Lewis Carroll

```
sea([N,M,L,R,I]) :-
        sat(M =< N),    % statement 1
        sat(L =< R),    % statement 2
        sat(I =< ~R),   % statement 3
        sat(N =< L).    % statement 4

implication_chain([], Prev) --> [Prev].
implication_chain(Vs0, Prev) --> [Prev],
        { select(V, Vs0, Vs) },
        (   { taut(Prev =< V, 1) } ->
            implication_chain(Vs, V)
        ;   { taut(Prev =< ~V, 1) } ->
            implication_chain(Vs, ~V)
        ).
```

Lewis Carroll

```
sea([N,M,L,R,I]) :-
        sat(M =< N),   % statement 1
        sat(L =< R),   % statement 2
        sat(I =< ~R),  % statement 3
        sat(N =< L).   % statement 4

implication_chain([], Prev) --> [Prev].
implication_chain(Vs0, Prev) --> [Prev],
        { select(V, Vs0, Vs) },
        (   { taut(Prev =< V, 1) } ->
            implication_chain(Vs, V)
        ;   { taut(Prev =< ~V, 1) } ->
            implication_chain(Vs, ~V)
        ).

?- sea(Vs),
   Vs = [N,M,L,R,I],
   select(Start, Vs, Rest),
   phrase(implication_chain(Rest, Start), Cs).
```

Lewis Carroll

```
sea([N,M,L,R,I]) :-
        sat(M =< N),    % statement 1
        sat(L =< R),    % statement 2
        sat(I =< ~R),   % statement 3
        sat(N =< L).    % statement 4

implication_chain([], Prev) --> [Prev].
implication_chain(Vs0, Prev) --> [Prev],
        { select(V, Vs0, Vs) },
        (   { taut(Prev =< V, 1) } ->
            implication_chain(Vs, V)
        ;   { taut(Prev =< ~V, 1) } ->
            implication_chain(Vs, ~V)
        ).

?- sea(Vs),
   Vs = [N,M,L,R,I],
   select(Start, Vs, Rest),
   phrase(implication_chain(Rest, Start), Cs).
```

The two solutions are `[M,N,L,R,~I]` and `[I,~R,~L,~N,~M]`.

Table of Contents

Cut

The *cut* (`!`) is a control operator that *commits* Prolog to decisions made up to that point – like `break` to some extent.

Cut

The *cut* (`!`) is a control operator that *commits* Prolog to decisions made up to that point – like `break` to some extent.

prunes choice points, preventing Prolog from backtracking past it.

Cut

The *cut* (`!`) is a control operator that *commits* Prolog to decisions made up to that point – like `break` to some extent.

prunes choice points, preventing Prolog from backtracking past it.

It can improve efficiency, but may break logical purity.

Cut

The *cut* (`!`) is a control operator that *commits* Prolog to decisions made up to that point – like `break` to some extent.

prunes choice points, preventing Prolog from backtracking past it.

It can improve efficiency, but may break logical purity.

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

Cut

The *cut* (`!`) is a control operator that *commits* Prolog to decisions made up to that point – like `break` to some extent.

prunes choice points, preventing Prolog from backtracking past it.

It can improve efficiency, but may break logical purity.

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

?- max(5, 3, M).
M = 5.

?- max(3, 5, M).
M = 5.

?- max(5, 3, 3).
false.
```

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program.

It prunes *correct* alternatives — and must be used with care.

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program.
It prunes *correct* alternatives — and must be used with care.

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

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program. It prunes *correct* alternatives — and must be used with care.

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

- If `X > Y` succeeds, cut commits to the first clause.
- Otherwise, Prolog tries the second clause.

```
?- max(10, 0, 0).
   true.
```

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program. It prunes *correct* alternatives — and must be used with care.

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

- If `X > Y` succeeds, cut commits to the first clause.
- Otherwise, Prolog tries the second clause.

```
?- max(10, 0, 0).
   true.
```

- Head `max(10, 0, 0)` matches second clause.
- Skips test `X < Y`, gives wrong answer.

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program. It prunes *correct* alternatives — and must be used with care.

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

- If `X > Y` succeeds, cut commits to the first clause.
- Otherwise, Prolog tries the second clause.

```
?- max(10, 0, 0).
   true.
```

- Head `max(10, 0, 0)` matches second clause.
- Skips test `X < Y`, gives wrong answer.

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

Red Cuts: Committing to a Clause

A *red cut* is a semantic cut that affects the logic of the program.
It prunes *correct* alternatives — and must be used with care.

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

- If `X > Y` succeeds, cut commits to the first clause.
- Otherwise, Prolog tries the second clause.

```
?- max(10, 0, 0).
  true.
```

- Head `max(10, 0, 0)` matches second clause.
- Skips test `X < Y`, gives wrong answer.

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

- Use cut *after* input test.
- Postpone output unification (`Z = X`) *until after the cut*.

Grue Cuts: Improving Efficiency Without Changing Logic

- A *grue cut* does *not affect logical correctness*.

Grue Cuts: Improving Efficiency Without Changing Logic

- A *grue cut* does *not affect logical correctness*.
- It is used to *prune redundant or irrelevant choices*, improving performance.

Grue Cuts: Improving Efficiency Without Changing Logic

- A *grue cut* does *not affect logical correctness*.
- It is used to *prune redundant or irrelevant choices*, improving performance.
- Grue cuts come in two types:
  - *Blue cuts*: notify Prolog of determinism it should have inferred.
  - *Green cuts*: discard unhelpful proof paths.

Grue Cuts: Improving Efficiency Without Changing Logic

- A *grue cut* does *not affect logical correctness*.
- It is used to *prune redundant or irrelevant choices*, improving performance.
- Grue cuts come in two types:
  - *Blue cuts*: notify Prolog of determinism it should have inferred.
  - *Green cuts*: discard unhelpful proof paths.

Blue cut example:

```
capital(britain, london) :- !.
capital(australia, canberra) :- !.
capital(new_zealand, wellington) :- !.
```

Guidelines for Using Cuts

- *Cut only when necessary.* Use cuts to enforce determinism, not to hide logical problems.
- *Encapsulate cuts.* Cuts should appear *within the predicate they affect*, not in callers.
- *Postpone output unification until after the cut.* Avoid premature binding that could lead to incorrect answers under backtracking.
- *Avoid multiple cuts per clause.* More than one cut often signals design issues. Refactor instead.
- *Don't replace proper design with cuts.* If a predicate should be determinate, make it so logically rather than relying on ‘¡.
- *Use `once/1` for clarity.* Prefer `once(Goal)` when you only need the first solution without full pruning.
- *Document red cuts clearly.* If a cut changes semantics, explain why it's safe and necessary.

Questions?