DTU Course 02156 Logical Systems and Logic Programming (2021)

Week	Date	Main Topics (Prolog Programming in All Lessons)
35 #01	31/8	Course Prerequisites & Tutorial on Logical Systems and Logic Programming
36 #02	7/9	Chapter 1 - Introduction (Prolog Note)
37 #03	14/9	Chapter 2 - Propositional Logic: Formulas, Models, Tableaux
38 #04	21/9	Chapter 3 - Propositional Logic: Deductive Systems
39 #05	28/9	"Isabelle" - Propositional Logic: Sequent Calculus Verifier (SeCaV)
40 #06	5/10	Chapter 4 - Propositional Logic: Resolution
41 #07	12/10	Chapter 7 - First-Order Logic: Formulas, Models, Tableaux
42		(Autumn Vacation)
43 #08	26/10	Chapter 8 - First-Order Logic: Deductive Systems
44 #09	2/11	"Isabelle" - First-Order Logic: Sequent Calculus Verifier (SeCaV)
45 #10	9/11	Chapter 9 - First-Order Logic: Terms and Normal Forms
46 #11	16/11	Chapter 10 - First-Order Logic: Resolution
47 #12	23/11	Chapter 11 - First-Order Logic: Logic Programming
48 #13	30/11	Chapter 12 - First-Order Logic: Undecidability and Model Theory & Course Evaluation

Responsible: Associate Professor Jørgen Villadsen <jovi@dtu.dk>

Assignments & Exam **MUST BE SOLVED INDIVIDUALLY**

Assignment-1 Deadline Sunday 26/9 (Available Wednesday 15/9)

Assignment-2 Deadline Sunday 10/10 (Available Wednesday 29/9)

Assignment-3 Deadline Sunday 31/10 (Available Wednesday 13/10)

Assignment-4 Deadline Sunday 14/11 (Available Wednesday 3/11)

Assignment-5 Deadline Thursday 2/12 (Available Wednesday 17/11)

Written Exam Tuesday 14/12 (2 Hours / No Computer / All Notes Allowed)

The mandatory assignments and the written exam are evaluated as a whole – even if you do well in the mandatory assignments then you still must do decent in the written exam in order to pass the course!

A TEACHER MUST IMMEDIATELY REPORT ANY SUSPICION OF CHEATING TO THE STUDY ADMINISTRATION FOR FURTHER ACTIONS

Queue as two stacks using list reverse (constant amortized complexity)

```
empty(aqueue([], [])).

enqueue(X, aqueue(Xs, Ys), aqueue([X|Xs], Ys)).

dequeue(Y, aqueue(Xs, [Y|Ys]), aqueue(Xs, Ys)).
dequeue(Y, aqueue(Xs, []), aqueue([], Ys)) :- reverse(Xs, [Y|Ys]).
```

```
? empty(Q0), enqueue(N, Q0, Q1), dequeue(N, Q1, Q2).

Q0 = Q2
Q1 = aqueue([N], [])
Q2 = aqueue([], [])

Yes
```

Haskell

```
import Prelude (Maybe(..), print, reverse)

data Queue a = AQueue [a] [a]

empty = AQueue [] []

enqueue x (AQueue xs ys) = AQueue (x : xs) ys

dequeue (AQueue [] []) = (Nothing, AQueue [] [])
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys)
dequeue (AQueue (x : xs) []) =
  (case reverse (x : xs) of y : ys -> (Just y, AQueue [] ys))

main = print (case dequeue (enqueue 0 empty) of (Just x, _) -> x)
```

Isabelle

```
theory Queue imports Main begin

datatype 'a queue = AQueue ‹'a list› ‹'a list›

definition empty :: ‹'a queue› where ‹empty ≡ AQueue [] []›

fun enqueue where
  ‹enqueue x (AQueue xs ys) = AQueue (x # xs) ys›

fun dequeue where
  ‹dequeue (AQueue [] []) = (None, AQueue [] [])› |
  ‹dequeue (AQueue xs (y # ys)) = (Some y, AQueue xs ys)› |
  ‹dequeue (AQueue (x # xs) []) = (case rev (x # xs) of y # ys ⇒ (Some y, AQueue [] ys))›

theorem ‹(case (dequeue (enqueue n empty)) of (Some x, _) ⇒ x) = n›
  unfolding empty_def by simp

export_code empty enqueue dequeue in Haskell

end
```

Info

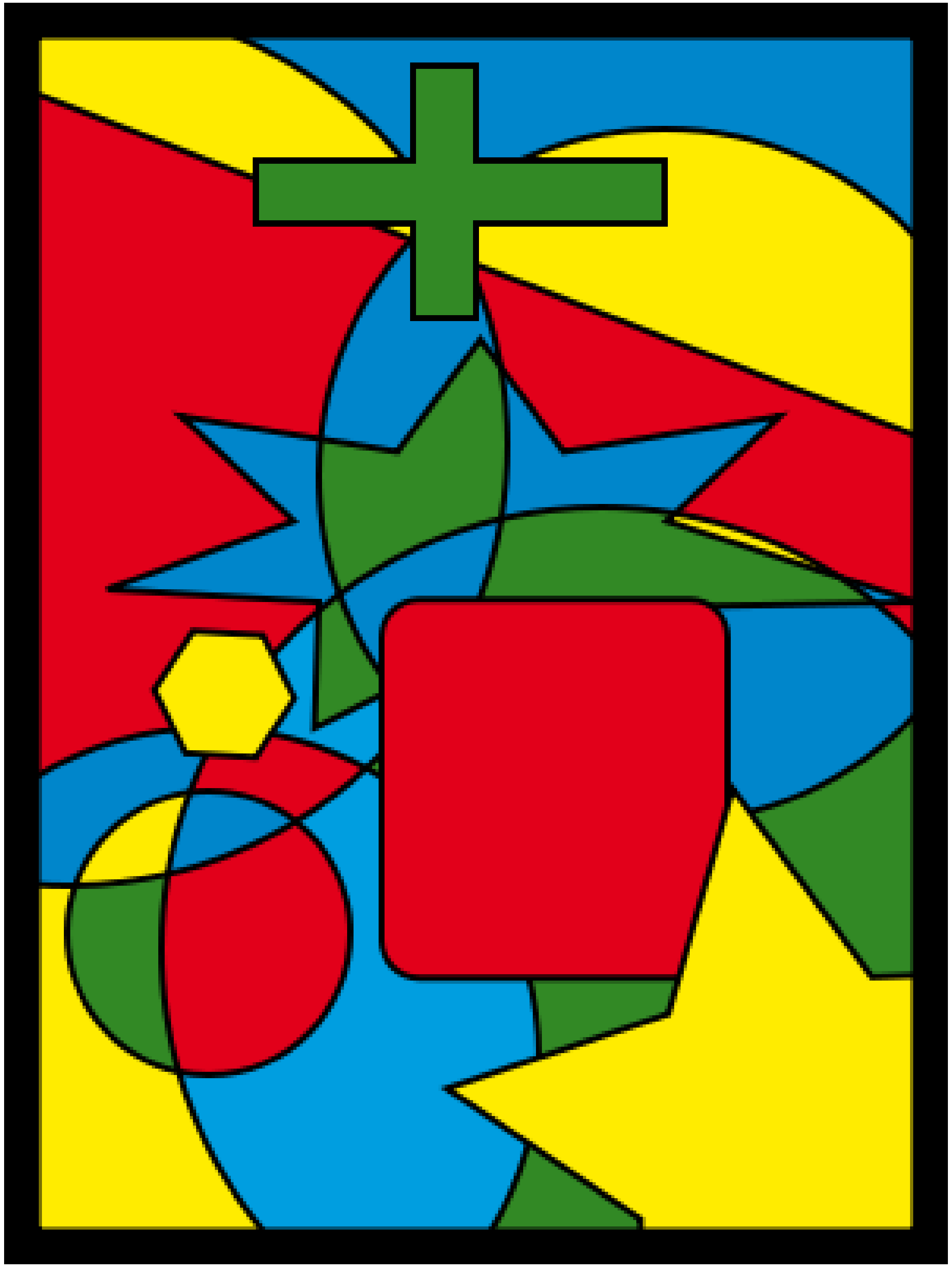
Search: 50%

```
Found termination order: "{}"
```

Info

Search: 50%

```
See theory exports
```



02156 Highlight Weeks 1-5

Important Prolog programs to study:

- Quicksort & Mergesort Algorithms
- Member & Append `basic.pl`
- Truth Table Checker `logic.pl`
- Map Colouring `map.pl`
- Resolution Prover `resolution.pl`

The Four Colour Theorem was the first major theorem to be proven using a computer…

Picture source:

http://commons.wikimedia.org/wiki/Image:Four_Colour_Map_Example.svg

Agenda — Week #4

The four colour theorem — Computer proof — Exercise

The tracer in SWI-Prolog

Prolog note — Cut (pages 10-11)

Propositional logic — Tableaux summary

Gentzen System

Hilbert System

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The problem whether four colours are enough was posed in 1852 and solved in 1976.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The problem whether four colours are enough was posed in 1852 and solved in 1976.

The four colour theorem was the first major theorem proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The problem whether four colours are enough was posed in 1852 and solved in 1976.

The four colour theorem was the first major theorem proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand.

Related to the four colour theorem is map colouring — namely given a map, find a colouring using at most four colours.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The problem whether four colours are enough was posed in 1852 and solved in 1976.

The four colour theorem was the first major theorem proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand.

Related to the four colour theorem is map colouring — namely given a map, find a colouring using at most four colours.

The four colour theorem states that this is always possible.

The Four Colour Theorem

The four colour theorem states that given any plane separated into regions, the regions may be coloured using at most four colours in such a way that no two adjacent regions get the same colour.

A plane separated into regions can be seen as a map of countries.

The problem whether four colours are enough was posed in 1852 and solved in 1976.

The four colour theorem was the first major theorem proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand.

Related to the four colour theorem is map colouring — namely given a map, find a colouring using at most four colours.

The four colour theorem states that this is always possible.

Today's final exercise is to do map colouring in Prolog... :-)

Map Colouring Exercise — Suggestion 1

Use numbers 0,1,2,3 as colours and show the colours of the neighbours as a list:

```
?- test(X).

X = [ (0, [1, 2, 1], austria),
      (0, [3, 2, 1], belgium),
      (0, [1], denmark),
      (3, [0, 1, 2, 0, 1], france),
      (1, [3, 0, 2, 2, 0, 0], germany),
      (2, [0, 1], holland),
      (1, [3, 0, 2], italy),
      (1, [0], portugal),
      (0, [3, 1], spain),
      (2, [3, 1, 0, 1], switzerland) ]

Yes
```

Map Colouring Exercise — Suggestion 2

In logic programming a variable like X is both input and output which is very different from functional programming!

```
test(X) :- map(X), colouring(X).

map([
  (A,[I,S,G],austria),
  (B,[F,H,G],belgium),
  (D,[G],denmark),
  (F,[E,I,S,B,G],france),
  (G,[F,A,S,H,B,D],germany),
  (H,[B,G],holland),
  (I,[F,A,S],italy),
  (P,[E],portugal),
  (E,[F,P],spain),
  (S,[F,I,A,G],switzerland)
]).
```

Map Colouring Exercise — Suggestion 3

Prolog is elegant but can be a bit hard to master — you must experiment...

Please ask me or the teaching assistants for help if needed!

If SWI-Prolog hides part of the answer behind `...` then enter `w` for `write` in order to have everything written — and enter `p` for `print` in order to have `...` printed again.

Basic Predicates

```
% Prolog file basic.pl

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).

append([],U,U).
append([H|T],U,[H|V]) :- append(T,U,V).
```

Tracer Example

```
member(H,[_|T]) :- member(H,T). % Note that the clauses are swapped
member(H,[H|_]).
```

Tracer Example

```
member(H,[_|T]) :- member(H,T). % Note that the clauses are swapped
member(H,[H|_]).

?- trace, member(X,[a,b]).
```

Tracer Example

```
member(H,[_|T]) :- member(H,T). % Note that the clauses are swapped
member(H,[H|_]).

?- trace, member(X,[a,b]).

   Call: (1) member(_0,[a,b]) ?
   Call: (2) member(_0,[b]) ?
   Call: (3) member(_0,[]) ?
   Fail: (3) member(_0,[]) ?
   Redo: (2) member(_0,[b]) ?
   Exit: (2) member(b,[b]) ?
   Exit: (1) member(b,[a,b]) ?

X = b ;

   Redo: (1) member(_0,[a,b]) ?
   Exit: (1) member(a,[a,b]) ?

X = a ;

No
```

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

The printed lines might differ slightly from the shown ones.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

The printed lines might differ slightly from the shown ones.

As shown the SWI-Prolog predicate `trace/0` enables the tracer (recall that the `0` means that the predicate takes no arguments).

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

The printed lines might differ slightly from the shown ones.

As shown the SWI-Prolog predicate `trace/0` enables the tracer (recall that the `0` means that the predicate takes no arguments).

Use the SWI-Prolog predicates `notrace/0` and `nodebug/0` to disable the tracer and the debugger, respectively.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

The printed lines might differ slightly from the shown ones.

As shown the SWI-Prolog predicate `trace/0` enables the tracer (recall that the `0` means that the predicate takes no arguments).

Use the SWI-Prolog predicates `notrace/0` and `nodebug/0` to disable the tracer and the debugger, respectively.

The debugger is not explained here, but it can stop at so-called spy points.

Tracer I

The built-in and library auto-loaded predicates are traced in a different way.

So load the file `basic.pl` with the predicates `member` and `append` if you want to trace them.

Otherwise the trace will be less informative.

The printed lines might differ slightly from the shown ones.

As shown the SWI-Prolog predicate `trace/0` enables the tracer (recall that the `0` means that the predicate takes no arguments).

Use the SWI-Prolog predicates `notrace/0` and `nodebug/0` to disable the tracer and the debugger, respectively.

The debugger is not explained here, but it can stop at so-called spy points.

See the SWI-Prolog reference manual for the details about the tracer and the debugger.

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

A unique call counter is shown in parentheses (the counter usually does not start at zero).

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

A unique call counter is shown in parentheses (the counter usually does not start at zero).

The variables in arguments have special (unique) names starting with _ (underscore).

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

A unique call counter is shown in parentheses (the counter usually does not start at zero).

The variables in arguments have special (unique) names starting with _ (underscore).

Enter a blank line at the tracer prompt `?` to continue to the next port (this is called creeping).

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

A unique call counter is shown in parentheses (the counter usually does not start at zero).

The variables in arguments have special (unique) names starting with _ (underscore).

Enter a blank line at the tracer prompt `?` to continue to the next port (this is called creeping).

Enter `s` to continue to the next port of the current predicate with the same call counter (thus skipping all ports even of the same predicate as long as the call counter is higher than the current call counter).

Tracer II

The tracer shows the port (`Call`, `Exit`, `Redo`, `Fail`) and the current predicate and arguments.

A unique call counter is shown in parentheses (the counter usually does not start at zero).

The variables in arguments have special (unique) names starting with _ (underscore).

Enter a blank line at the tracer prompt `?` to continue to the next port (this is called creeping).

Enter `s` to continue to the next port of the current predicate with the same call counter (thus skipping all ports even of the same predicate as long as the call counter is higher than the current call counter).

Enter `a` to abort the query.

Tracer III

Many other tracer commands are possible, but creeping, skipping, and aborting are the most useful for small programs.

Tracer III

Many other tracer commands are possible, but creeping, skipping, and aborting are the most useful for small programs.

It is optional to use the tracer for the exercises and for the assignments too.

Key Prolog Programs — Examples

```
?- list([1,2,3]).

Yes
```

```
?- length([1,2,3],_).

Yes
```

```
?- append([1,2,3],_,_).

Yes
```

```
?- member(2,[1,2,3]).

Yes
```

```
?- select(2,[1,2,3],_).

Yes
```

Key Prolog Programs — More Examples

```
?- select(X,[1,2,3],L).

X = 1
L = [2, 3] ;

X = 2
L = [1, 3] ;

X = 3
L = [1, 2] ;

No
```

Key Prolog Programs — Comparison

```
list([]).
list([_|T]) :- list(T).

length([],0).
length([_|T],N1) :- length(T,N), N1 is N+1.

append([],U,U).
append([H|T],U,[H|V]) :- append(T,U,V).

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).

select(X,[X|T],T).
select(X,[Y|T],[Y|U]) :- select(X,T,U).
```

Cut

Prolog has a special nullary predicate called *cut* with symbol: `!`

Cut

Prolog has a special nullary predicate called *cut* with symbol: `!`

It has nothing to do with the factorial function.

Cut

Prolog has a special nullary predicate called *cut* with symbol: `!`

It has nothing to do with the factorial function.

As illustrated in the following example the cut discards alternatives.

Cut

Prolog has a special nullary predicate called *cut* with symbol: `!`

It has nothing to do with the factorial function.

As illustrated in the following example the cut discards alternatives.

Original version of the partition program for Quicksort:

```
part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs) :- X > Y, part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs]) :- X =< Y, part(X,Xs,Ls,Bs).
```

Cut

Prolog has a special nullary predicate called *cut* with symbol: `!`

It has nothing to do with the factorial function.

As illustrated in the following example the cut discards alternatives.

Original version of the partition program for Quicksort:

```
part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs) :- X > Y, part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs]) :- X =< Y, part(X,Xs,Ls,Bs).
```

Slightly more efficient version:

```
part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs) :- X > Y, !, part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs]) :- part(X,Xs,Ls,Bs).
```

Cut — Definition

Consider the following schematic clauses for a predicate `p`:

$$
\begin{array}{l}
\texttt{p(}\mathbf{s}_1\texttt{) :- } \mathbf{A}_1. \\
\ldots \\
\texttt{p(}\mathbf{s}_i\texttt{) :- } \mathbf{B}\texttt{,!,}\mathbf{C}. \\
\ldots \\
\texttt{p(}\mathbf{s}_k\texttt{) :- } \mathbf{A}_k.
\end{array}
$$

Suppose that during the execution of a query a call $\texttt{p(}\mathbf{t}\texttt{)}$ is encountered and eventually the *i*-th clause is used and the indicated occurrence of the cut is executed.

Cut — Definition

Consider the following schematic clauses for a predicate p:

```
p(s_1)  :-  A_1.
...
p(s_i)  :-  B,!,C.
...
p(s_k)  :-  A_k.
```

Suppose that during the execution of a query a call p(**t**) is encountered and eventually the i-th clause is used and the indicated occurrence of the cut is executed.

Then the indicated occurrence of the cut succeeds immediately, but additionally all alternative ways of computing **B** are discarded, and all computations of p(**t**) using the $i + 1$-th to k-th clause for p are discarded as backtrackable alternatives to the current selection of the i-clause.

Cut — Example — Motivation

Consider the basic `member` predicate:

```
member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
```

Recall that it can be used to enumerate the elements of the list.

Cut — Example — Motivation

Consider the basic `member` predicate:

```
member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
```

Recall that it can be used to enumerate the elements of the list.

It can also succeed more than one time when used to check whether an element is a member of a given list:

```
?- member(a,[a,b,a,c]), write(x), fail.
xx

No
```

The choice point left by the non-deterministic `member` can cause problems in certain situations.

Cut — Example — Implementation

The `membercheck` predicate works as the basic predicate `member`, except that it is deterministic and never leaves a choice point:

```
?- membercheck(a,[a,b,a,c]), write(x), fail.
x

No
```

A cut is used in the first clause in order to commit to the first match and avoid backtracking:

```
membercheck(H,[H|_]) :- !.
membercheck(H,[_|T]) :- membercheck(H,T).
```

Use the more efficient `membercheck` in preference to the basic `member`, but only where its restrictions are appropriate.

Cut — Example — Implementation

The `membercheck` predicate works as the basic predicate `member`, except that it is deterministic and never leaves a choice point:

```
?- membercheck(a,[a,b,a,c]), write(x), fail.
x

No
```

A cut is used in the first clause in order to commit to the first match and avoid backtracking:

```
membercheck(H,[H|_]) :- !.
membercheck(H,[_|T]) :- membercheck(H,T).
```

Use the more efficient `membercheck` in preference to the basic `member`, but only where its restrictions are appropriate.

In SWI-Prolog a built-in predicate `memberchk` is available.

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

- Tableaux
- Axiomatics
- Resolution

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

- Tableaux
- Axiomatics
- Resolution

Provability: $\vdash A$ iff there is a proof in the particular axiomatics (system $\mathcal{G}$, $\mathcal{H}$, ...)

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

- Tableaux
- Axiomatics
- Resolution

Provability: $\vdash A$ iff there is a proof in the particular axiomatics (system $\mathcal{G}$, $\mathcal{H}$, ...)

Soundness and Completeness Theorem: $\models A$ iff $\vdash A$

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

- Tableaux
- Axiomatics
- Resolution

Provability: $\vdash A$ iff there is a proof in the particular axiomatics (system $\mathcal{G}$, $\mathcal{H}$, ...)

Soundness and Completeness Theorem: $\models A$ **iff** $\vdash A$

Similar theorems for tableaux and resolution.

Propositional Logic

Validity: $\models A$ iff it is true for all interpretations

- Tableaux
- Axiomatics
- Resolution

Provability: $\vdash A$ iff there is a proof in the particular axiomatics (system $\mathcal{G}$, $\mathcal{H}$, ...)

Soundness and Completeness Theorem: $\models A$ iff $\vdash A$

Similar theorems for tableaux and resolution.

Much more about tableaux, axiomatics and resolution soon for first-order logic. :-)

Different Tableaux 1

The formula (p => q) => ((r \/ p) => (r \/ q)) is valid because the following tableau is closed:

```
~((p => q) => ((r \/ p) => (r \/ q)))
   (p => q),~((r \/ p) => (r \/ q))
---
L R
      ~p,~((r \/ p) => (r \/ q))
         (r \/ p),~(r \/ q),~p
-----
LL LR
            r,~(r \/ q),~p
               ~r,~q,~p,r
X
LR
            p,~(r \/ q),~p
               ~r,~q,~p,p
X
R
      q,~((r \/ p) => (r \/ q))
         (r \/ p),~(r \/ q),q
-----
RL RR
            r,~(r \/ q),q
               ~r,~q,q,r
X
RR
            p,~(r \/ q),q
               ~r,~q,q,p
X
```

Different Tableaux 2

```
The formula (p => q) => ((r \/ p) => (r \/ q)) is valid because the following tableau is closed:

~((p => q) => ((r \/ p) => (r \/ q)))
   ~((r \/ p) => (r \/ q)),(p => q)
      (r \/ p),~(r \/ q),(p => q)
---
L R
         r,~(r \/ q),(p => q)
            ~r,~q,(p => q),r
-----
LL LR
               ~p,r,~r,~q
X
LR
               q,r,~r,~q
X
R
         p,~(r \/ q),(p => q)
            ~r,~q,(p => q),p
-----
RL RR
               ~p,p,~r,~q
X
RR
               q,p,~r,~q
X
```

Different Tableaux 3

```
The formula (p => q) => ((r \/ p) => (r \/ q)) is valid because the following tableau is closed:

~((p => q) => ((r \/ p) => (r \/ q)))
   ~((r \/ p) => (r \/ q)),(p => q)
      ~(r \/ q),(r \/ p),(p => q)
        ~r,~q,(r \/ p),(p => q)
---
L R
            r,(p => q),~r,~q
-----
LL LR
               ~p,~r,~q,r
X
LR
               q,~r,~q,r
X
R
            p,(p => q),~r,~q
-----
RL RR
               ~p,~r,~q,p
X
RR
               q,~r,~q,p
X
```

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

If all branches of the completed tableaux are marked closed, then A is valid.

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

If all branches of the completed tableaux are marked closed, then A is valid.

A literal is an atom or a negation of an atom.

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

If all branches of the completed tableaux are marked closed, then A is valid.

A literal is an atom or a negation of an atom.

A leaf consisting of literals and with a complementary pair of literals $(p, \neg p)$ for some p is marked closed: $\times$

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

If all branches of the completed tableaux are marked closed, then A is valid.

A literal is an atom or a negation of an atom.

A leaf consisting of literals and with a complementary pair of literals $(p, \neg p)$ for some p is marked closed: $\times$

A leaf consisting of literals and without a complementary pair of literals $(p, \neg p)$ for some p is marked open: $\odot$

Tableaux

For deciding validity of a formula A, start the tableau with $\neg A$.

If all branches of the completed tableaux are marked closed, then A is valid.

A literal is an atom or a negation of an atom.

A leaf consisting of literals and with a complementary pair of literals $(p, \neg p)$ for some p is marked closed: $\times$

A leaf consisting of literals and without a complementary pair of literals $(p, \neg p)$ for some p is marked open: $\odot$

Two types of rules for tableaux: α-rules create a single child and β-rules create two children.

The Gentzen System

Tableaux turned "tree upside down and signs reversed":

$\neg((p \vee q) \rightarrow (q \vee p))$

$p \vee q, \neg(q \vee p)$

$p \vee q, \neg q, \neg p$

$\overline{\text{L} \qquad \text{R}}$

$p, \neg q, \neg p$

$\times$

R

$q, \neg q, \neg p$

$\times$

1.	$\vdash \neg p, q, p$	Axiom
2.	$\vdash \neg q, q, p$	Axiom
3.	$\vdash \neg(p \vee q), q, p$	$\beta\vee$, 1, 2
4.	$\vdash \neg(p \vee q), (q \vee p)$	$\alpha\vee$, 3
5.	$\vdash (p \vee q) \rightarrow (q \vee p)$	$\alpha \rightarrow$, 4

The Gentzen System

Tableaux turned “tree upside down and signs reversed”:

$\neg((p \vee q) \to (q \vee p))$
$p \vee q, \neg(q \vee p)$
$p \vee q, \neg q, \neg p$

$\overline{\text{L} \qquad \text{R}}$

L

$p, \neg q, \neg p$
$\times$

R

$q, \neg q, \neg p$
$\times$

1.	$\vdash \neg p, q, p$	Axiom
2.	$\vdash \neg q, q, p$	Axiom
3.	$\vdash \neg(p \vee q), q, p$	$\beta\vee$, 1, 2
4.	$\vdash \neg(p \vee q), (q \vee p)$	$\alpha\vee$, 3
5.	$\vdash (p \vee q) \to (q \vee p)$	$\alpha \to$, 4

Hence $\vdash (A \vee B) \to (B \vee A)$

The Gentzen System

Tableaux turned "tree upside down and signs reversed":

$\neg((p \vee q) \rightarrow (q \vee p))$
$p \vee q, \neg(q \vee p)$
$p \vee q, \neg q, \neg p$

L R

$p, \neg q, \neg p$
$\times$

R

$q, \neg q, \neg p$
$\times$

1.	$\vdash \neg p, q, p$	Axiom
2.	$\vdash \neg q, q, p$	Axiom
3.	$\vdash \neg(p \vee q), q, p$	$\beta\vee$, 1, 2
4.	$\vdash \neg(p \vee q), (q \vee p)$	$\alpha\vee$, 3
5.	$\vdash (p \vee q) \rightarrow (q \vee p)$	$\alpha \rightarrow$, 4

Hence $\vdash (A \vee B) \rightarrow (B \vee A)$

Soundness and Completeness Theorem: $\models A$ **iff** $\vdash A$

The Gentzen System

Tableaux turned "tree upside down and signs reversed":

$\neg((p \vee q) \rightarrow (q \vee p))$
$p \vee q, \neg(q \vee p)$
$p \vee q, \neg q, \neg p$

L R

$p, \neg q, \neg p$
$\times$

R

$q, \neg q, \neg p$
$\times$

1.	$\vdash \neg p, q, p$	Axiom
2.	$\vdash \neg q, q, p$	Axiom
3.	$\vdash \neg(p \vee q), q, p$	$\beta\vee$, 1, 2
4.	$\vdash \neg(p \vee q), (q \vee p)$	$\alpha\vee$, 3
5.	$\vdash (p \vee q) \rightarrow (q \vee p)$	$\alpha \rightarrow$, 4

Hence $\vdash (A \vee B) \rightarrow (B \vee A)$

Soundness and Completeness Theorem: $\models A$ **iff** $\vdash A$

Note: Use implicit set formation for conclusions in the system $\mathcal{G}$

The Hilbert System

One rule MP: $\vdash A, \vdash A \to B \ / \vdash B$

The Hilbert System

One rule MP: $\vdash A$, $\vdash A \to B \ / \vdash B$

Axiom 1: $\vdash A \to B \to A$

The Hilbert System

One rule MP: $\vdash A, \vdash A \to B \;/\; \vdash B$

Axiom 1: $\vdash A \to B \to A$

Axiom 2: $\vdash (A \to B \to C) \to (A \to B) \to A \to C$

The Hilbert System

One rule MP: $\vdash A, \vdash A \to B \;/\; \vdash B$

Axiom 1: $\vdash A \to B \to A$

Axiom 2: $\vdash (A \to B \to C) \to (A \to B) \to A \to C$

Axiom 3: $\vdash (\neg B \to \neg A) \to A \to B$

The Hilbert System

One rule MP: $\vdash A, \vdash A \to B \;/ \vdash B$

Axiom 1: $\vdash A \to B \to A$

Axiom 2: $\vdash (A \to B \to C) \to (A \to B) \to A \to C$

Axiom 3: $\vdash (\neg B \to \neg A) \to A \to B$

Complicated even for the following tiny example.

1.	$\vdash (A \to (A \to A) \to A) \to (A \to A \to A) \to A \to A$	Axiom 2
2.	$\vdash A \to (A \to A) \to A$	Axiom 1
3.	$\vdash (A \to A \to A) \to A \to A$	MP 1, 2
4.	$\vdash A \to A \to A$	Axiom 1
5.	$\vdash A \to A$	MP 3, 4

The Hilbert System

One rule MP: $\vdash A, \vdash A \to B \;/\; \vdash B$

Axiom 1: $\vdash A \to B \to A$

Axiom 2: $\vdash (A \to B \to C) \to (A \to B) \to A \to C$

Axiom 3: $\vdash (\neg B \to \neg A) \to A \to B$

Complicated even for the following tiny example.

1.	$\vdash (A \to (A \to A) \to A) \to (A \to A \to A) \to A \to A$	Axiom 2
2.	$\vdash A \to (A \to A) \to A$	Axiom 1
3.	$\vdash (A \to A \to A) \to A \to A$	MP 1, 2
4.	$\vdash A \to A \to A$	Axiom 1
5.	$\vdash A \to A$	MP 3, 4

Soundness and Completeness Theorem: $\models A$ **iff** $\vdash A$

The Hilbert System

One rule MP: $\vdash A, \vdash A \rightarrow B \;/ \vdash B$

Axiom 1: $\vdash A \rightarrow B \rightarrow A$

Axiom 2: $\vdash (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$

Axiom 3: $\vdash (\neg B \rightarrow \neg A) \rightarrow A \rightarrow B$

Complicated even for the following tiny example.

1.	$\vdash (A \rightarrow (A \rightarrow A) \rightarrow A) \rightarrow (A \rightarrow A \rightarrow A) \rightarrow A \rightarrow A$	Axiom 2
2.	$\vdash A \rightarrow (A \rightarrow A) \rightarrow A$	Axiom 1
3.	$\vdash (A \rightarrow A \rightarrow A) \rightarrow A \rightarrow A$	MP 1, 2
4.	$\vdash A \rightarrow A \rightarrow A$	Axiom 1
5.	$\vdash A \rightarrow A$	MP 3, 4

Soundness and Completeness Theorem: $\models A$ **iff** $\vdash A$

Note: Use implicit set formation for assumptions in system $\mathcal{H}$

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

Assumption rule: $U \vdash A_i$ ($A_i \in U$)

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

Assumption rule: $U \vdash A_i$ $(A_i \in U)$

Deduction rule: $U \cup \{A\} \vdash B$ / $U \vdash A \to B$

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

Assumption rule: $U \vdash A_i$ $(A_i \in U)$

Deduction rule: $U \cup \{A\} \vdash B$ / $U \vdash A \rightarrow B$

Deduction theorem: The deduction rule is a sound derived rule

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

Assumption rule: $U \vdash A_i$ ($A_i \in U$)

Deduction rule: $U \cup \{A\} \vdash B \;/\; U \vdash A \rightarrow B$

Deduction theorem: The deduction rule is a sound derived rule

Axiom 3 is not necessary in order to do without the deduction rule.

The Deduction Rule

Let U be a set of formulas and let $U \vdash A$ equal $\vdash A$ iff $U = \emptyset$

Assumption rule: $U \vdash A_i$ $(A_i \in U)$

Deduction rule: $U \cup \{A\} \vdash B \;/\; U \vdash A \to B$

Deduction theorem: The deduction rule is a sound derived rule

Axiom 3 is not necessary in order to do without the deduction rule.

Classical logic is explosive — See theorems 3.20 and 3.21.

1.	$\{\neg A\} \vdash \neg A \to \neg B \to \neg A$	Axiom 1
2.	$\{\neg A\} \vdash \neg A$	Assumption
3.	$\{\neg A\} \vdash \neg B \to \neg A$	MP 1, 2
4.	$\{\neg A\} \vdash (\neg B \to \neg A) \to (A \to B)$	Axiom 3
5.	$\{\neg A\} \vdash A \to B$	MP 3, 4
6.	$\vdash \neg A \to A \to B$	Deduction 5

Non-explosive logics are also called paraconsistent logics.

Resolution

After the tableaux, the Gentzen system and the Hilbert system then resolution is investigated and implemented.

Resolution

After the tableaux, the Gentzen system and the Hilbert system then resolution is investigated and implemented.

John Alan Robinson:
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM, 12(1):23–41, 1965.

Resolution

After the tableaux, the Gentzen system and the Hilbert system then resolution is investigated and implemented.

John Alan Robinson:
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM, 12(1):23–41, 1965.

Soundness and completeness must be established in all cases.