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

Agenda — Week #6

Motivation — Logic Programming & Prolog — IBM Watson :-)

Cut summary

Prolog note — Negation-As-Failure etc. (pages 11-13)

Occurs-Check

Communications of the ACM...

Resolution

Motivation — Logic Programming & Prolog

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Visual Prolog — Prolog Development Center (Denmark)

http://www.visual-prolog.com

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Visual Prolog — Prolog Development Center (Denmark)

`http://www.visual-prolog.com`

P# — A concurrent Prolog for .NET

`http://homepages.inf.ed.ac.uk/stg/research/Psharp`

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Visual Prolog — Prolog Development Center (Denmark)

`http://www.visual-prolog.com`

P# — A concurrent Prolog for .NET

`http://homepages.inf.ed.ac.uk/stg/research/Psharp`

λProlog — The typed λ-calculus (Higher-Order Programming)

`http://www.lix.polytechnique.fr/~dale/lProlog`

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Visual Prolog — Prolog Development Center (Denmark)

`http://www.visual-prolog.com`

P# — A concurrent Prolog for .NET

`http://homepages.inf.ed.ac.uk/stg/research/Psharp`

λProlog — The typed λ-calculus (Higher-Order Programming)

`http://www.lix.polytechnique.fr/~dale/lProlog`

Mercury is a new, purely declarative logic programming language

`http://www.mercurylang.org`

Motivation — Logic Programming & Prolog

A list of a few of the many languages and systems available...

Visual Prolog — Prolog Development Center (Denmark)

`http://www.visual-prolog.com`

P# — A concurrent Prolog for .NET

`http://homepages.inf.ed.ac.uk/stg/research/Psharp`

λProlog — The typed λ-calculus (Higher-Order Programming)

`http://www.lix.polytechnique.fr/~dale/lProlog`

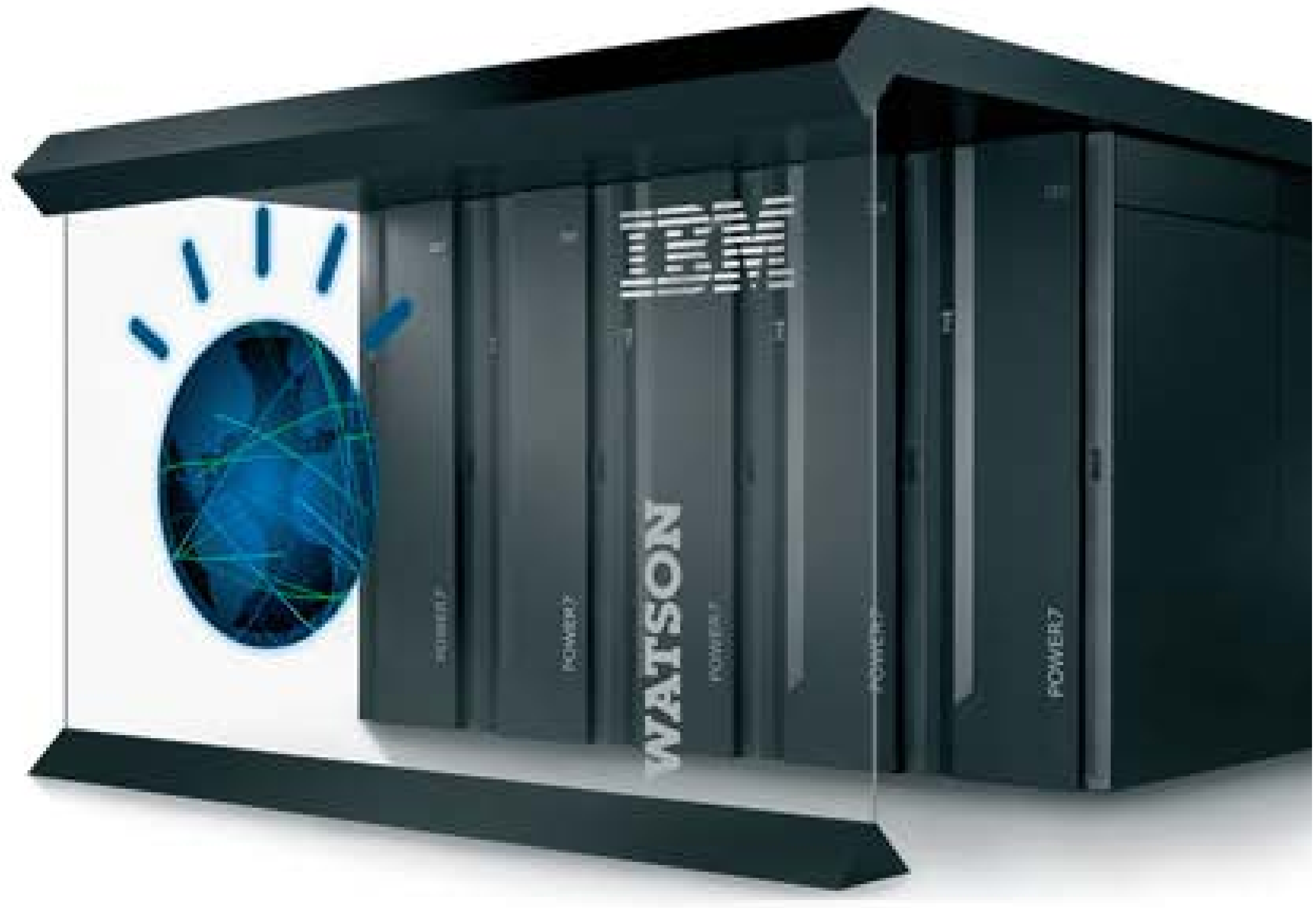
Mercury is a new, purely declarative logic programming language

`http://www.mercurylang.org`

But ISO Prolog is also widely used...

Watson is IBM’s AI computer capable of answering natural language questions

Watson was named after IBM's first president, Thomas J. Watson (1874-1956)



This Is Watson – IBM Journal of Research and Development – Vol. 56, No. 3/4, May/July 2012

In 2007, IBM Research took on the grand challenge of building a computer system that could compete with champions at the game of *Jeopardy!*.

In 2011, the open-domain question-answering system dubbed Watson beat the two highest ranked players in a nationally televised two-game *Jeopardy!* match.

This special issue provides a deep technical overview of the ideas and accomplishments that positioned our team to take on the *Jeopardy!* challenge, build Watson, and ultimately triumph.

It describes the nature of the question-answering challenge represented by *Jeopardy!* and details our technical approach.

The papers herein describe and provide experimental results for many of the algorithmic techniques developed as part of the Watson system, covering areas including computational linguistics, information retrieval, knowledge representation and reasoning, and machine learning.

The papers offer component-level evaluations as well as their end-to-end contribution to Watson's overall question-answering performance.

Summary:

- **The research team consisted of about 25 full-time researchers and engineers.**
- **Watson is powered by 10 racks of IBM Power 750 servers running Linux.**
- **Watson has 2,880 processor cores and 15 terabytes of random access memory.**
- **The programming was done mostly in Java but also significant chunks in C++ and Prolog.**

Quotes:

“Most of our rule-based question analysis components are implemented in Prolog, a well-established standard for representing pattern-matching rules.”

“Our implementation can analyze a question in a fraction of a second, which is necessary to be competitive at the *Jeopardy!* task.”

“We found that Prolog was the ideal choice for the language because of its simplicity and expressiveness.”

“Using Prolog for this task has significantly improved our productivity in developing new pattern-matching rules and has delivered the execution efficiency necessary to be competitive in a *Jeopardy!* game.”

References:

http://www.ibmwatson.com/

http://www.youtube.com/watch?v=lI-M7O_bRNg

http://ieeexplore.ieee.org/xpl/tocresult.jsp?reload=true&isnumber=6177717

http://asmarterplanet.com/blog/2011/02/the-watson-research-team-answers-your-questions.html

http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/

Cut Again 1

The cut `!` always succeeds but cannot be backtracked past.

Cut Again 1

The cut ! always succeeds but cannot be backtracked past.

It avoids additional computations that are not desired or required.

```
p(X) :- a(X).
p(X) :- b(X), !, c(X).
p(X) :- d(X).

a(1). a(2). b(1). b(2). c(1). c(2). d(1). d(2).
```

The cut should be used sparingly and as early as possible.

Cut Again 1

The cut `!` always succeeds but cannot be backtracked past.

It avoids additional computations that are not desired or required.

```
p(X) :- a(X).
p(X) :- b(X), !, c(X).
p(X) :- d(X).
```

```
a(1). a(2). b(1). b(2). c(1). c(2). d(1). d(2).
```

The cut should be used sparingly and as early as possible.

Never just insert cuts into code that is not working correctly.

Cut Again 1

The cut `!` always succeeds but cannot be backtracked past.

It avoids additional computations that are not desired or required.

```
p(X) :- a(X).
p(X) :- b(X), !, c(X).
p(X) :- d(X).

a(1). a(2). b(1). b(2). c(1). c(2). d(1). d(2).
```

The cut should be used sparingly and as early as possible.

Never just insert cuts into code that is not working correctly.

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

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

Cut Again 2

Consider the use of lists as sets (no duplicate elements in the lists).

```
?- intersection([a,b,c],[b,d],Z).

Z = [b] ;

No

?- union([a,b,c],[b,d],Z).

Z = [a, c, b, d] ;

No
```

In case the basic predicate `member` is used, would it be appropriate to use the deterministic predicate `membercheck` instead?

Cut Again 3

Yes, `membercheck` is appropriate, since due to the cut after `member` no choice point is needed.

```
intersection([],_,[]) :- !.
intersection([H|T],L,I) :-
  membercheck(H,L), !, I = [H|R], intersection(T,L,R).
intersection([_|T],L,R) :-
  intersection(T,L,R).
```

The cut in the first clause is not needed for the given instantiation pattern `intersection(+Set1,+Set2,?Set3)`, but it makes the program deterministic for the most liberal instantiation pattern `intersection(?Set1,?Set2,?Set3)`.

Cut Again 3

Yes, `membercheck` is appropriate, since due to the cut after `member` no choice point is needed.

```
intersection([],_,[]) :- !.
intersection([H|T],L,I) :-
  membercheck(H,L), !, I = [H|R], intersection(T,L,R).
intersection([_|T],L,R) :-
  intersection(T,L,R).
```

The cut in the first clause is not needed for the given instantiation pattern `intersection(+Set1,+Set2,?Set3)`, but it makes the program deterministic for the most liberal instantiation pattern `intersection(?Set1,?Set2,?Set3)`.

It is important that `I = [H|R]` is after the cut in the second clause, otherwise the following query will succeed: `intersection([a],[a],[])`.

Cut Again 4

Yes, `membercheck` is appropriate, since due to the cut after `member` no choice point is needed.

```
union([],L,L) :- !.
union([H|T],L,R) :-
  membercheck(H,L), !, union(T,L,R).
union([H|T],L,[H|R]) :-
  union(T,L,R).
```

The cut in the first clause is not needed for the given instantiation pattern `union(+Set1,+Set2,?Set3)`, but it makes the program deterministic for the most liberal instantiation pattern `union(?Set1,?Set2,?Set3)`.

Cut Again 4

Yes, `membercheck` is appropriate, since due to the cut after `member` no choice point is needed.

```
union([],L,L) :- !.
union([H|T],L,R) :-
  membercheck(H,L), !, union(T,L,R).
union([H|T],L,[H|R]) :-
  union(T,L,R).
```

The cut in the first clause is not needed for the given instantiation pattern `union(+Set1,+Set2,?Set3)`, but it makes the program deterministic for the most liberal instantiation pattern `union(?Set1,?Set2,?Set3)`.

SWI-Prolog has the predicates as library auto-loaded.

Cut Again 4

Yes, `membercheck` is appropriate, since due to the cut after `member` no choice point is needed.

```
union([],L,L) :- !.
union([H|T],L,R) :-
  membercheck(H,L), !, union(T,L,R).
union([H|T],L,[H|R]) :-
  union(T,L,R).
```

The cut in the first clause is not needed for the given instantiation pattern `union(+Set1,+Set2,?Set3)`, but it makes the program deterministic for the most liberal instantiation pattern `union(?Set1,?Set2,?Set3)`.

SWI-Prolog has the predicates as library auto-loaded.

And note that using logical variables it can still be tail-recursive.

Negation-As-Failure

The prefix operator \+ is the so-called *negation-as-failure* operator (non-provability).

Negation-As-Failure

The prefix operator \+ is the so-called *negation-as-failure* operator (non-provability).

The infix predicate \= ("not-equal") is defined as follows:

```
X \= Y :-
  \+ X = Y.
```

The predicate \= succeeds iff the arguments are not equal in the sense that they do not unify, for example:

```
?- a \= b.

Yes

?- a \= a.

No
```

Negation-As-Failure — Warning

Care must be taken with uninstantiated variables:

```
?- X \= Y.

No

?- X \= X.

No
```

Recall that unlike the arithmetic equality `=:=` and non-equality `=\=` predicates the = and \= predicates do not evaluate the arguments:

```
?- 2+2 \= 4.

Yes
```

Negation-As-Failure — Definition

The use of `\+ p` for a given predicate `p` (possibly with arguments) is equivalent to a new predicate `not_p` (with the same arguments) with the following definition (again with the same arguments in both clauses):

```
not_p :- p, !, fail.
not_p.
```

Hence the definition of the infix predicate `\=` is equivalent to the following clauses (with anonymous variables in the second clause):

```
X \= Y :- X = Y, !, fail.
_ \= _.
```

If-Then-Else

Prolog has a special *if-then-else* construction: `B -> S ; T`

If-Then-Else

Prolog has a special *if-then-else* construction: `B -> S ; T`

If `B` succeeds then the result is `S` and otherwise the result is `T` (choice points in `B` are discarded).

If-Then-Else

Prolog has a special *if-then-else* construction: `B -> S ; T`

If `B` succeeds then the result is `S` and otherwise the result is `T` (choice points in `B` are discarded).

For example consider a program `add(+Elem,+List1,?List2)` that succeeds iff adding `Elem` once to `List1` gives `List2` (assuming that it is not to be added if already there).

```
add(X,Y,Z) :- member(X,Y), !, Z = Y.
add(X,Y,Z) :- Z = [X|Y].
```

Alternative:

```
add(X,Y,Z) :- member(X,Y) -> Z = Y ; Z = [X|Y].
```

Would it be appropriate to use the deterministic `membercheck` predicate instead?

If-Then-Else — Another Example

Various versions 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).
```

```
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).
```

```
part(_,[],[],[]).
part(X,[Y|Xs],Ls,Bs) :-
  ( X > Y ->
    Ls = [Y|L1s], part(X,Xs,L1s,Bs)
  ;
    Bs = [Y|B1s], part(X,Xs,Ls,B1s)
  ).
```

If-Then-Else — Warning

Nested use of if-then-else is possible, but it does not always give more understandable logic programs.

```
part(X,X1s,Ls,Bs) :-
  ( X1s = [] ->
    Ls = [], Bs = []
  ;
    X1s = [Y|Xs],
    ( X > Y ->
      Ls = [Y|L1s], part(X,Xs,L1s,Bs)
    ;
      Bs = [Y|B1s], part(X,Xs,Ls,B1s)
    )
  ).
```

Often the result is close to functional programs.

The Special So-Called Univ Predicate

Instantiation patterns `+Term =.. ?List` and `-Term =.. +List` (it is an error if both `List` and `Term` are variables).

The Special So-Called Univ Predicate

Instantiation patterns `+Term =.. ?List` and `-Term =.. +List` (it is an error if both `List` and `Term` are variables).

`List` is a list which head is the functor of `Term` and which tail is a list of the arguments of the term.

```
?- baz(hello, X) =.. List.

List = [baz, hello, X]

Yes

?- Term =.. [foo, bar(1)]

Term = foo(bar(1))

Yes
```

Occurs-Check

For efficiency reasons the so-called occurs-check is omitted in Prolog:

```
?- A = f(A).

A = f(A)

Yes
```

```
?- A = f(g(h(A))).

A = f(g(h(A)))

Yes
```

Occurs-Check

For efficiency reasons the so-called occurs-check is omitted in Prolog:

```
?- A = f(A).

A = f(A)

Yes
```

```
?- A = f(g(h(A))).

A = f(g(h(A)))

Yes
```

This is usually no problem.

Occurs-Check

For efficiency reasons the so-called occurs-check is omitted in Prolog:

```
?- A = f(A).

A = f(A)

Yes
```

```
?- A = f(g(h(A))).

A = f(g(h(A)))

Yes
```

This is usually no problem.

But observe the following situations:

```
?- unify_with_occurs_check(A,f(A)).

No
```

```
?- A \= f(A).

No
```

Communications of the ACM...

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

http://cacm.acm.org/magazines/2009/8

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

http://cacm.acm.org/magazines/2009/8

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

http://cacm.acm.org/magazines/2009/8

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications is recognized as the most trusted and knowledgeable source of industry information for today's computing professionals.

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

`http://cacm.acm.org/magazines/2009/8`

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications is recognized as the most trusted and knowledgeable source of industry information for today's computing professionals.

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as NP-complete.

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

http://cacm.acm.org/magazines/2009/8

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications is recognized as the most trusted and knowledgeable source of industry information for today's computing professionals.

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as NP-complete.

An instance of any one of these problems can be relatively easily transformed into an instance of another.

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

`http://cacm.acm.org/magazines/2009/8`

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications is recognized as the most trusted and knowledgeable source of industry information for today's computing professionals.

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as NP-complete.

An instance of any one of these problems can be relatively easily transformed into an instance of another.

Whether there exist subexponential solutions to NP-Complete problems is arguably the most famous open question in computer science.

Communications of the ACM...

Boolean Satisfiability:
From Theoretical Hardness to Practical Success

http://cacm.acm.org/magazines/2009/8

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and a profession.

Communications is recognized as the most trusted and knowledgeable source of industry information for today's computing professionals.

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as NP-complete.

An instance of any one of these problems can be relatively easily transformed into an instance of another.

Whether there exist subexponential solutions to NP-Complete problems is arguably the most famous open question in computer science.

The success with SAT has led to its widespread commercial use in certain domains such as design and verification of hardware and software systems.

Axiomatics

Recall the Hilbert system with rule MP: $\vdash A, \vdash A \rightarrow B \;/\; \vdash B$

Axiomatics

Recall the Hilbert system with rule MP: $\vdash A, \vdash A \to B \ / \vdash B$

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

Axiomatics

Recall the Hilbert system with rule MP: $\vdash A, \vdash A \to B \ / \vdash B$

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

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

Axiomatics

Recall the Hilbert system with 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': $\vdash (\neg B \to \neg A) \to (\neg B \to A) \to B$

Axiomatics

Recall the Hilbert system with 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': $\vdash (\neg B \rightarrow \neg A) \rightarrow (\neg B \rightarrow A) \rightarrow B$

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

Axiomatics

Recall the Hilbert system with 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': $\vdash (\neg B \rightarrow \neg A) \rightarrow (\neg B \rightarrow A) \rightarrow B$

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

Usually a Hilbert system will need the derived deduction rule to allow for substantial proofs.

Example

$(p \to q) \lor (q \to p)$

Example

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

```
?- truthtable( (p => q) \ (q => p) ).
(p=>q)\ (q=>p)  p q  value
                t t    t
                t f    t
                f t    t
                f f    t


Yes
```

Example

$(p \to q) \lor (q \to p)$

```
?- truthtable( (p => q) \ (q => p) ).
(p=>q)\ (q=>p)  p q  value
                t t    t
                t f    t
                f t    t
                f f    t


Yes
```

Valid means true for all interpretations: $\models (p \to q) \lor (q \to p)$

Example — Gentzen System

Use rules similar to the rules for tableaux:

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

Example — Gentzen System

Use rules similar to the rules for tableaux:

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

In fact always construct the tableaux first (explicitly or implicitly).

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

~((p => q) \ (q => p))
   ~(p => q),~(q => p)
      p,~q,~(q => p)
         q,~p,p,~q
X
```

Example — Hilbert System

MP: $U \vdash A$, $U \vdash A \rightarrow B$ / $U \vdash B$

$U \vdash A$ if A is an axiom.

Example — Hilbert System

MP: $U \vdash A$, $U \vdash A \to B$ / $U \vdash B$

$U \vdash A$ if A is an axiom.

So for any A if $\vdash A$ then $U \vdash A$.

Example — Hilbert System

MP: $U \vdash A$, $U \vdash A \to B$ / $U \vdash B$ $\qquad$ $U \vdash A$ if A is an axiom.

So for any A if $\vdash A$ then $U \vdash A$.

Implicit set formation used for assumptions:

1.	$q, \neg p, p \vdash q$	Assumption
2.	$q, \neg p \vdash p \to q$	Deduction 1
3.	$q, \neg p \vdash \neg\neg(p \to q)$	Double negation 2
4.	$q \vdash \neg p \to \neg\neg(p \to q)$	Deduction 3
5.	$q \vdash \neg(p \to q) \to p$	Contrapositive 4
6.	$\vdash q \to \neg(p \to q) \to p$	Deduction 5
7.	$\vdash \neg(p \to q) \to q \to p$	Exchange of antecedent 6
8.	$\vdash (p \to q) \vee (q \to p)$	Def. of $\vee$

Theorem 3.18: $\vdash (A \to B \to C) \to B \to A \to C$

Example — Hilbert System

MP: $U \vdash A$, $U \vdash A \to B$ / $U \vdash B$ $\quad$ $U \vdash A$ if A is an axiom.

So for any A if $\vdash A$ then $U \vdash A$.

Implicit set formation used for assumptions:

1.	$q, \neg p, p \vdash q$	Assumption
2.	$q, \neg p \vdash p \to q$	Deduction 1
3.	$q, \neg p \vdash \neg\neg(p \to q)$	Double negation 2
4.	$q \vdash \neg p \to \neg\neg(p \to q)$	Deduction 3
5.	$q \vdash \neg(p \to q) \to p$	Contrapositive 4
6.	$\vdash q \to \neg(p \to q) \to p$	Deduction 5
7.	$\vdash \neg(p \to q) \to q \to p$	Exchange of antecedent 6
8.	$\vdash (p \to q) \vee (q \to p)$	Def. of $\vee$

Theorem 3.18: $\vdash (A \to B \to C) \to B \to A \to C$

Any theorem of the form $U \vdash A \to B$ justifies a rule of the form $U \vdash A$ / $U \vdash B$ simply by using MP.

Example — Resolution

CNF transformation: $\neg((p \rightarrow q) \vee (q \rightarrow p)) \;\equiv\; p \wedge \neg q \wedge q \wedge \neg p$

Example — Resolution

CNF transformation: $\neg((p \rightarrow q) \vee (q \rightarrow p)) \equiv p \wedge \neg q \wedge q \wedge \neg p$

$$
\begin{array}{lll}
1. & p & \\
2. & \overline{q} & \\
3. & q & \\
4. & \overline{p} & \\
5. & \square & 1,4
\end{array}
$$

Example — Resolution

CNF transformation: $\neg((p \rightarrow q) \vee (q \rightarrow p)) \equiv p \wedge \neg q \wedge q \wedge \neg p$

1. p
2. $\overline{q}$
3. q
4. $\overline{p}$
5. $\square$ 1,4

```
?- resolution( (p => q) \ (q => p) ).
~ ((p=>q)\ (q=>p))
(p& ~q)&q& ~p
[[p], [neg q], [q], [neg p]]
[[], [p], [neg q], [q], [neg p]]

Yes
```

Propositional Resolution

$$\neg((p \rightarrow q) \vee (q \rightarrow p)) \;\equiv\; \neg(\neg p \vee q) \wedge \neg(\neg q \vee p) \;\equiv\; p \wedge \neg q \wedge q \wedge \neg p$$

Propositional Resolution

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

1.	p	
2.	$\overline{q}$	
3.	q	
4.	$\overline{p}$	
5.	$\square$	1,4

Propositional Resolution

$$\neg((p \to q) \vee (q \to p)) \;\equiv\; \neg(\neg p \vee q) \wedge \neg(\neg q \vee p) \;\equiv\; p \wedge \neg q \wedge q \wedge \neg p$$

$$\begin{array}{lll} 1. & p & \\ 2. & \overline{q} & \\ 3. & q & \\ 4. & \overline{p} & \\ 5. & \square & 1,4 \end{array}$$

$$A \to B \;\equiv\; \neg A \vee B \qquad\qquad A \leftrightarrow B \;\equiv\; (A \to B) \wedge (B \to A)$$

Propositional Resolution

$$\neg((p \rightarrow q) \vee (q \rightarrow p)) \;\equiv\; \neg(\neg p \vee q) \wedge \neg(\neg q \vee p) \;\equiv\; p \wedge \neg q \wedge q \wedge \neg p$$

1.	p	
2.	$\overline{q}$	
3.	q	
4.	$\overline{p}$	
5.	$\square$	1,4

$$A \rightarrow B \quad \equiv \quad \neg A \vee B \qquad\qquad A \leftrightarrow B \quad \equiv \quad (A \rightarrow B) \wedge (B \rightarrow A)$$

$$\neg(A \wedge B) \quad \equiv \quad (\neg A \vee \neg B) \qquad\qquad \neg(A \vee B) \quad \equiv \quad (\neg A \wedge \neg B)$$

Propositional Resolution

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

1.	p	
2.	$\overline{q}$	
3.	q	
4.	$\overline{p}$	
5.	$\Box$	1,4

$$A \rightarrow B \equiv \neg A \vee B \qquad A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B) \qquad \neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

$$\neg\neg A \equiv A$$

Propositional Resolution

$$\neg((p \to q) \lor (q \to p)) \equiv \neg(\neg p \lor q) \land \neg(\neg q \lor p) \equiv p \land \neg q \land q \land \neg p$$

1. p
2. $\overline{q}$
3. q
4. $\overline{p}$
5. $\square$ 1,4

$$A \to B \equiv \neg A \lor B \qquad A \leftrightarrow B \equiv (A \to B) \land (B \to A)$$

$$\neg(A \land B) \equiv (\neg A \lor \neg B) \qquad \neg(A \lor B) \equiv (\neg A \land \neg B)$$

$$\neg\neg A \equiv A$$

$$A \lor (B \land C) \equiv (A \lor B) \land (A \lor C)$$
$$(A \land B) \lor C \equiv (A \lor C) \land (B \lor C)$$

CNF

A formula is in *conjunctive normal form* (CNF) iff it is a conjunction of disjunctions of literals.

CNF

A formula is in *conjunctive normal form* (CNF) iff it is a conjunction of disjunctions of literals.

Every formula in propositional logic can be transformed into an equivalent formula in CNF:

$$
\begin{array}{rl}
 & (\neg p \to \neg q) \to p \to q \\
\equiv & \neg(\neg\neg p \vee \neg q) \vee \neg p \vee q \\
\equiv & (\neg\neg\neg p \wedge \neg\neg q) \vee \neg p \vee q \\
\equiv & (\neg p \wedge q) \vee \neg p \vee q \\
\equiv & (\neg p \vee \neg p \vee q) \wedge (q \vee \neg p \vee q) \\
\equiv & (\neg p \vee q) \wedge (q \vee \neg p) \\
\equiv & (\neg p \vee q)
\end{array}
$$

Using $\to$ elimination, De Morgan's laws to push $\neg$ inward, double negation elimination, distribution of $\vee$ over $\wedge$, and commutativity, associativity and idempotence of disjunction and conjunction.

Clauses

A *clause* is a set of literals which is considered to be an implicit disjunction.

Clauses

A *clause* is a set of literals which is considered to be an implicit disjunction.

A formula in *clausal form* is a set of clauses which is considered to be an implicit conjunction.

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

$$\{\{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$$

$$\{\overline{q}\,\overline{p}\,q, p\,\overline{p}\,q\}$$

If ℓ is a literal then ℓ^c is its complement (p to $\overline{p}$ and vice versa).

Clauses

A *clause* is a set of literals which is considered to be an implicit disjunction.

A formula in *clausal form* is a set of clauses which is considered to be an implicit conjunction.

$$(\neg q \lor \neg p \lor q) \land (p \lor \neg p \lor q \lor p \lor \neg p)$$

$$\{\{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$$

$$\{\overline{q}\,\overline{p}\,q, p\,\overline{p}\,q\}$$

If ℓ is a literal then ℓ^c is its complement (p to $\overline{p}$ and vice versa).

The empty clause $\square$ is unsatisfiable (the set of clauses $\emptyset$ is valid).

Clauses

A *clause* is a set of literals which is considered to be an implicit disjunction.

A formula in *clausal form* is a set of clauses which is considered to be an implicit conjunction.

$$(\neg q \lor \neg p \lor q) \land (p \lor \neg p \lor q \lor p \lor \neg p)$$

$$\{\{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$$

$$\{\overline{q}\,\overline{p}\,q, p\,\overline{p}\,q\}$$

If ℓ is a literal then ℓ^c is its complement (p to $\overline{p}$ and vice versa).

The empty clause $\square$ is unsatisfiable (the set of clauses $\emptyset$ is valid).

The resolution rule maintains satisfiability, hence if $\square$ is obtained then the original set of clauses must have been unsatisfiable.

Clauses

A *clause* is a set of literals which is considered to be an implicit disjunction.

A formula in *clausal form* is a set of clauses which is considered to be an implicit conjunction.

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

$$\{\{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$$

$$\{\overline{q}\,\overline{p}\,q, p\,\overline{p}\,q\}$$

If ℓ is a literal then ℓ^c is its complement (p to $\overline{p}$ and vice versa).

The empty clause $\Box$ is unsatisfiable (the set of clauses $\emptyset$ is valid).

The resolution rule maintains satisfiability, hence if $\Box$ is obtained then the original set of clauses must have been unsatisfiable.

To prove A, derive $\Box$ from $\neg A$ in clausal form (refutation).

Resolution Program

CNF transformation / Clausal form:

```
~((p => q => r) => (p => q) => p => r)     Eliminate =>
~(~(~p \ ~q \ r) \ ~(~p \ q) \ ~p \ r)     Push ~ inwards
(~p \ ~q \ r) & (~p \ q) & p & ~r          Already in CNF
[~p,~q,r],[~p,q],[p],[~r]                  Clausal form
```

Resolution Program

CNF transformation / Clausal form:

```
~((p => q => r) => (p => q) => p => r)     Eliminate =>
~(~(~p \ ~q \ r) \ ~(~p \ q) \ ~p \ r)     Push ~ inwards
(~p \ ~q \ r) & (~p \ q) & p & ~r          Already in CNF
[~p,~q,r],[~p,q],[p],[~r]                  Clausal form
```

About 100 lines of code in file `resolution.pl` :-)

Resolution Program

CNF transformation / Clausal form:

```
~((p => q => r) => (p => q) => p => r)     Eliminate =>
~(~(~p \ ~q \ r) \ ~(~p \ q) \ ~p \ r)     Push ~ inwards
(~p \ ~q \ r) & (~p \ q) & p & ~r          Already in CNF
[~p,~q,r],[~p,q],[p],[~r]                  Clausal form
```

About 100 lines of code in file `resolution.pl` :-)

```
?- resolution(((p => (q => r)) => ((p => q) => (p => r)))).
~ ((p=>q=>r)=> (p=>q)=>p=>r)
(~p\ ~q\r)& (~p\q)&p& ~r
[[neg p,neg q,r],[neg p,q],[p],[neg r]]
[[r,neg p],[neg p,neg q,r],[neg p,q],[p],[neg r]]
[[r],[r,neg p],[neg p,neg q,r],[neg p,q],[p],[neg r]]
[[],[r],[r,neg p],[neg p,neg q,r],[neg p,q],[p],[neg r]]

Yes
```

Resolution

Resolution rule: $C_1,\ C_2\ /\ (C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1,\ \ell^c \in C_2)$

Resolution

Resolution rule: $C_1,\ C_2\ /\ (C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1,\ \ell^c \in C_2)$

The clauses C_1, C_2 are called *clashing clauses* (they clash on ℓ, ℓ^c) and are parent clauses of the child clause, the *resolvent clause*.

1.	$\overline{p}\,\overline{q}\,r$	
2.	$\overline{p}\,q$	
3.	p	
4.	$\overline{r}$	
5.	$\overline{p}\,\overline{q}$	4,1
6.	$\overline{p}$	5,2
7.	$\square$	6,3

Hence $(p \to q \to r) \to (p \to q) \to p \to r$ is proved.

Resolution

Resolution rule: $C_1,\ C_2\ /\ (C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1,\ \ell^c \in C_2)$

The clauses C_1, C_2 are called *clashing clauses* (they clash on ℓ, ℓ^c) and are parent clauses of the child clause, the *resolvent clause*.

1.	$\overline{p}\,\overline{q}\,r$	
2.	$\overline{p}\,q$	
3.	p	
4.	$\overline{r}$	
5.	$\overline{p}\,\overline{q}$	4,1
6.	$\overline{p}$	5,2
7.	$\square$	6,3

Hence $(p \to q \to r) \to (p \to q) \to p \to r$ is proved.

Resolution is sound and complete.