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

Did SWI-Prolog give up on ISO compliance?

Not (much) more than it used to. If you are looking for a Prolog system that restricts you to the ISO standard, SWI-Prolog should not be the first thing to look at. **ISO compliant programs** *that do not explore corner cases such as relying on specific behaviour on basically invalid programs (e.g., expecting length(42,X) to fail silently)* **should run fine**.

http://www.swi-prolog.org/Directions.txt

SWI-Prolog Manual: length(?List, ?Int)

True if Int represents the number of elements in List. This predicate is a true relation and can be used to find the length of a list or produce a list (holding variables) of length Int. The predicate is non-deterministic, producing lists of increasing length if List is a partial list and Int is unbound. It raises errors if

Int is bound to a non-integer.

Int is a negative integer.

List is neither a list nor a partial list. This error condition includes cyclic lists. *

This predicate fails if the tail of List is equivalent to Int (e.g., length(L,L)).

---

* ISO demands failure here. We think an error is more appropriate.

Agenda — Week #3

Prolog summary

Prolog note (pages 9-10 top)

Propositional logic — Tableaux

Basic Prolog Predicates

`member(?Elem,?List)` succeeds iff `Elem` unifies with one of the members of `List`.

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

For example `member(b,[a,b,c])` succeeds.

Basic Prolog Predicates

`member(?Elem,?List)` succeeds iff `Elem` unifies with one of the members of `List`.

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

For example `member(b,[a,b,c])` succeeds.

`append(?List1,?List2,?List3)` succeeds iff `List3` unifies with the concatenation of `List1` and `List2`.

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

For example `append([a,b],[c,d],[a,b,c,d])` succeeds.

Basic Prolog Predicates

`member(?Elem,?List)` succeeds iff `Elem` unifies with one of the members of `List`.

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

For example `member(b,[a,b,c])` succeeds.

`append(?List1,?List2,?List3)` succeeds iff `List3` unifies with the concatenation of `List1` and `List2`.

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

For example `append([a,b],[c,d],[a,b,c,d])` succeeds.

The predicates are not in ISO Prolog.

Tracer Example

```
?- trace, append(_,[X,X|_],[a,b,b,a,a,c]).
```

Tracer Example

```
?- trace, append(_,[X,X|_],[a,b,b,a,a,c]).
   Call: (1) append(_3,[_1,_1|_2],[a,b,b,a,a,c]) ?
   Redo: (1) append(_3,[_1,_1|_2],[a,b,b,a,a,c]) ?
   Call: (2) append(_4,[_1,_1|_2],[b,b,a,a,c]) ?
   Exit: (2) append([],[b,b,a,a,c],[b,b,a,a,c]) ?
   Exit: (1) append([a],[b,b,a,a,c],[a,b,b,a,a,c]) ?

X = b ;

   Redo: (2) append(_4,[_1,_1|_2],[b,b,a,a,c]) ?
   Call: (3) append(_5,[_1,_1|_2],[b,a,a,c]) ?
   Redo: (3) append(_5,[_1,_1|_2],[b,a,a,c]) ?
   Call: (4) append(_6,[_1,_1|_2],[a,a,c]) ?
   Exit: (4) append([],[a,a,c],[a,a,c]) ?
   Exit: (3) append([b],[a,a,c],[b,a,a,c]) ?
   Exit: (2) append([b,b],[a,a,c],[b,b,a,a,c]) ?
   Exit: (1) append([a,b,b],[a,a,c],[a,b,b,a,a,c]) ?
```

Tracer Example

```
X = a ;

   Redo: (4) append(_6,[_1,_1|_2],[a,a,c]) ?
   Call: (5) append(_7,[_1,_1|_2],[a,c]) ?
   Redo: (5) append(_7,[_1,_1|_2],[a,c]) ?
   Call: (6) append(_8,[_1,_1|_2],[c]) ?
   Redo: (6) append(_8,[_1,_1|_2],[c]) ?
   Call: (7) append(_9,[_1,_1|_2],[]) ?
   Redo: (7) append(_9,[_1,_1|_2],[]) ?
   Fail: (7) append(_9,[_1,_1|_2],[]) ?
   Fail: (6) append(_8,[_1,_1|_2],[c]) ?
   Fail: (5) append(_7,[_1,_1|_2],[a,c]) ?
   Fail: (4) append(_6,[_1,_1|_2],[a,a,c]) ?
   Fail: (3) append(_5,[_1,_1|_2],[b,a,a,c]) ?
   Fail: (2) append(_4,[_1,_1|_2],[b,b,a,a,c]) ?
   Fail: (1) append(_3,[_1,_1|_2],[a,b,b,a,a,c]) ?

No
```

Special Exercise

Special Exercise

For each of the following queries state the number of solutions and the answer substitutions.

```
?- member(X,[7,5,9,2,8,4,1,3,6]).
?- member(3,[7,5,X,2,8,Y,1,3,6]).
?- append([1,2,3],[a],Z).
?- append([1,2,3],Y,Z).
?- append(X,[a],Z).
?- append(a,a,[]).
?- append([],a,a).
?- member(a,L), member(b,L).
?- length(L,2), member(a,L), member(b,L).
?- length(L,3), member(a,L), member(b,L).
?- L=[_,_,_], member(a,L), member(b,L), append(_,[c,_],L).
?- length([X],X).
```

Explain the results.

Special Exercise

```
?- member(X,[7,5,9,2,8,4,1,3,6]).

X = 7 ;
X = 5 ;
X = 9 ;
X = 2 ;
X = 8 ;
X = 4 ;
X = 1 ;
X = 3 ;
X = 6 ;

No
```

Special Exercise

```
?- member(3,[7,5,X,2,8,Y,1,3,6]).

X = 3 ;
Y = 3 ;

Yes
```

Special Exercise

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

Z = [1, 2, 3, a] ;

No
```

Special Exercise

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

Z = [1, 2, 3|Y] ;

No
```

Special Exercise

```
?- append(X,[a],Z).

X = []
Z = [a] ;

X = [_1]
Z = [_1, a] ;

X = [_1, _2]
Z = [_1, _2, a] ;

X = [_1, _2, _3]
Z = [_1, _2, _3, a]

...
```

Special Exercise

```
?- append(a,a,[]).

No
```

Special Exercise

```
?- append([],a,a).

Yes
```

Special Exercise

```
?- member(a,L), member(b,L).

L = [a, b|_1] ;

L = [a, _1, b|_2] ;

L = [a, _1, _2, b|_3]

...
```

Special Exercise

```
?- length(L,2), member(a,L), member(b,L).

L = [a, b] ;

L = [b, a] ;

No
```

Special Exercise

```
?- length(L,3), member(a,L), member(b,L).

L = [a, b, _1] ;

L = [a, _1, b] ;

L = [b, a, _1] ;

L = [_1, a, b] ;

L = [b, _1, a] ;

L = [_1, b, a] ;

No
```

Special Exercise

```
?- L=[_,_,_], member(a,L), member(b,L), append(_,[c,_],L).

L = [a, c, b] ;

L = [b, c, a] ;

No
```

Special Exercise

```
?- length([X],X).

X = 1 ;

No
```

Unification

Unification with occurs-check only unifies a variable with a term if this term does not contain the variable itself...

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

A = f(A)

Yes

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

No
```

Much more later about unification and occurs-check!

SWI-Prolog Issue

The program `list(?List)` assumes a reasonable `length` program implementation!

```
list(L) :- length(L,_).
```

SWI-Prolog Issue

The program `list(?List)` assumes a reasonable `length` program implementation!

```
list(L) :- length(L,_).
```

For example:

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

But in SWI-Prolog `length([a|b],N)` gives an error!

SWI-Prolog Issue

The program `list(?List)` assumes a reasonable `length` program implementation!

```
list(L) :- length(L,_).
```

For example:

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

But in SWI-Prolog `length([a|b],N)` gives an error!

In YAP-Prolog and GNU-Prolog the result is OK:

```
?- length([a|b],N).

No
```

Directives

Directives are annotations for the compiler usually inserted in programs on separate lines with `:-` in front:

```
:- ensure_loaded(main).
```

```
:- op(400,yfx,xor).
```

Here are short descriptions:

`ensure_loaded(+File)` loads the `File` if is not already loaded (to load is to execute the directives and compile the programs).

`op(+Priority,+Specifier,+Name)` declares `Name` to be an operator of given `Specifier` (one of `fx`, `fy`, `xf`, `yf`, `yfx`, `xfy`, and `xfx` specifying prefix, postfix, infix, and associativity combinations where `x`-forms give no associativity) and `Priority` (a number 0–1200 where 1 is highest priority; 0 cancels the declaration).

Associativity

Example: Division usually associates to the left such that 3 / 4 / 5 is equivalent to (3 / 4) / 5 which denotes 0.15, unlike 3 / (4 / 5) which denotes 3.75.

Associativity

Example: Division usually associates to the left such that 3 / 4 / 5 is equivalent to (3 / 4) / 5 which denotes 0.15, unlike 3 / (4 / 5) which denotes 3.75.

The op predicate is for Prolog experts, but a basic understanding of operators is necessary.

```
:- op(400,yfx,xor).
```

Here it provides terms of the following equivalent forms:

```
P xor Q xor R
(P xor Q) xor R
xor(P,Q) xor R              xor(xor(P,Q),R)
```

Associativity — Examples

Specifier	Type	Associativity	Name - examples
fx	prefix	no	:-
fy	prefix	yes	+ -
xf	postfix	no	
yf	postfix	yes	
xfx	infix	no	:-
yfx	infix	left	+ - * /
xfy	infix	right	, ;

Operators

Priority	Specifier	Operators
1200	xfx	`:-  -->`
1200	fx	`:-`
1100	xfy	`;`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`= \=`
		`=.. == \== @< @=< @> @>=`
		`is =:= =\=  <  =<  >  >=`
600	xfy	`:`
500	yfx	`+ - /\ \/`
400	yfx	`* / // rem mod << >>`
200	xfy	`** ^`
200	fy	`+ - \`

See Exercises

See Exercises

Consider the formulas:

$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$

$(p \to q) \to p$

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

Is it possible to omit some of the parentheses?

Are any of the formulas satisfiable or valid?

See Exercises

Parentheses can be omitted from the first formula:

$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$

$(p \to (q \to r)) \to (p \to q) \to (p \to r)$

$(p \to (q \to r)) \to (p \to q) \to p \to r$

$(p \to q \to r) \to (p \to q) \to p \to r$

Parentheses cannot be omitted from the other formulas:

$(p \to q) \to p$

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

See Exercises

$(p \to q \to r) \to (p \to q) \to p \to r$

. . . .

```
(p=>q=>r)=> (p=>q)=>p=>r  p q r  value
                          t t t    t
                          t t f    t
                          t f t    t
                          t f f    t
                          f t t    t
                          f t f    t
                          f f t    t
                          f f f    t
```

. . . .

It is valid (and hence also satisfiable).

See Exercises

$(p \to q) \to p$

. . . .

```
(p=>q)=>p  p q  value
           t t    t
           t f    t
           f t    f
           f f    f
```

. . . .

It is not valid, but it is satisfiable.

See Exercises

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

. . . .

```
((p=>q)=>p)=>p  p q  value
                t t    t
                t f    t
                f t    t
                f f    t
```

. . . .

It is valid (and hence also satisfiable).

It is known as Peirce's law.

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

It is easier, but still hard, if the formulas are limited to propositional logic (propositional calculus) where there are no constants, variables, and quantifiers.

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

It is easier, but still hard, if the formulas are limited to propositional logic (propositional calculus) where there are no constants, variables, and quantifiers.

The textbook presents propositional logic as a “stepping stone” to FOL.

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

It is easier, but still hard, if the formulas are limited to propositional logic (propositional calculus) where there are no constants, variables, and quantifiers.

The textbook presents propositional logic as a "stepping stone" to FOL.

Propositional *formulas* are defined as follows ($\uparrow$, $\downarrow$, and $\oplus$ omitted):

$$\mathit{fml} ::= p \mid \neg \mathit{fml} \mid \mathit{fml} \wedge \mathit{fml} \mid \mathit{fml} \vee \mathit{fml} \mid \mathit{fml} \rightarrow \mathit{fml} \mid \mathit{fml} \leftrightarrow \mathit{fml}$$

The boolean type has values T and F.

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

It is easier, but still hard, if the formulas are limited to propositional logic (propositional calculus) where there are no constants, variables, and quantifiers.

The textbook presents propositional logic as a “stepping stone” to FOL.

Propositional *formulas* are defined as follows ($\uparrow$, $\downarrow$, and $\oplus$ omitted):

$$\mathit{fml} ::= p \mid \neg \mathit{fml} \mid \mathit{fml} \wedge \mathit{fml} \mid \mathit{fml} \vee \mathit{fml} \mid \mathit{fml} \rightarrow \mathit{fml} \mid \mathit{fml} \leftrightarrow \mathit{fml}$$

The boolean type has values T and F.

Remember the tables for the operators.

Propositional Logic

It is not easy for programs to produce formal proofs of given formulas in FOL (First-Order Logic).

It is easier, but still hard, if the formulas are limited to propositional logic (propositional calculus) where there are no constants, variables, and quantifiers.

The textbook presents propositional logic as a "stepping stone" to FOL.

Propositional *formulas* are defined as follows ($\uparrow$, $\downarrow$, and $\oplus$ omitted):

$$\mathit{fml} ::= p \mid \neg \mathit{fml} \mid \mathit{fml} \land \mathit{fml} \mid \mathit{fml} \lor \mathit{fml} \mid \mathit{fml} \rightarrow \mathit{fml} \mid \mathit{fml} \leftrightarrow \mathit{fml}$$

The boolean type has values T and F.

Remember the tables for the operators.

The set of operators is redundant (need only, say, $\neg$ and $\rightarrow$).

Interpretation

Set of atomic propositions, or *atoms*, $\mathcal{P}$ (propositional letters).

Interpretation

Set of atomic propositions, or *atoms*, $\mathcal{P}$ (propositional letters).

Set of formulas $\mathcal{F}$ generated by the grammar.

Interpretation

Set of atomic propositions, or *atoms*, $\mathcal{P}$ (propositional letters).

Set of formulas $\mathcal{F}$ generated by the grammar.

An *interpretation* is a function $\mathcal{I}\colon \mathcal{P} \mapsto \{T, F\}$.

Interpretation

Set of atomic propositions, or *atoms*, $\mathcal{P}$ (propositional letters).

Set of formulas $\mathcal{F}$ generated by the grammar.

An *interpretation* is a function $\mathcal{I} \colon \mathcal{P} \mapsto \{T, F\}$.

An interpretation $\mathcal{I}$ gives a truth value $v \colon \mathcal{F} \mapsto \{T, F\}$ given the semantics of the operators, for example:

$$\begin{aligned} v(\neg A_1) = F \quad &\text{iff} \quad v(A_1) = T \\ v(A_1 \rightarrow A_2) = F \quad &\text{iff} \quad v(A_1) = T \text{ and } v(A_2) = F \end{aligned}$$

Logical *equivalence* $A_1 \equiv A_2$ iff $v(A_1) = v(A_2)$ for all interpretations.

Interpretation

Set of atomic propositions, or *atoms*, $\mathcal{P}$ (propositional letters).

Set of formulas $\mathcal{F}$ generated by the grammar.

An *interpretation* is a function $\mathcal{I} \colon \mathcal{P} \mapsto \{T, F\}$.

An interpretation $\mathcal{I}$ gives a truth value $v \colon \mathcal{F} \mapsto \{T, F\}$ given the semantics of the operators, for example:

$$
\begin{array}{rcl}
v(\neg A_1) = F & \text{iff} & v(A_1) = T \\
v(A_1 \rightarrow A_2) = F & \text{iff} & v(A_1) = T \text{ and } v(A_2) = F
\end{array}
$$

Logical *equivalence* $A_1 \equiv A_2$ iff $v(A_1) = v(A_2)$ for all interpretations.

Generalized De Morgan's laws:

$$
\begin{array}{rcl}
\neg(A_1 \wedge \ldots \wedge A_n) & \equiv & \neg A_1 \vee \ldots \vee \neg A_n \\
\neg(A_1 \vee \ldots \vee A_n) & \equiv & \neg A_1 \wedge \ldots \wedge \neg A_n
\end{array}
$$

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations.

A valid propositional formula is also called a *tautology*.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations.

A valid propositional formula is also called a *tautology*.

A is *unsatisfiable* iff it is not satisfiable, that is if $v(A) = F$ for *all* interpretations.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations.

A valid propositional formula is also called a *tautology*.

A is *unsatisfiable* iff it is not satisfiable, that is if $v(A) = F$ for *all* interpretations.

Theorem: A is valid iff $\neg A$ is unsatisfiable.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations.

A valid propositional formula is also called a *tautology*.

A is *unsatisfiable* iff it is not satisfiable, that is if $v(A) = F$ for *all* interpretations.

Theorem: A is valid iff $\neg A$ is unsatisfiable.

Since any formula A contains a finite number of atoms, there is a finite number of different interpretations and these can be checked in order to decide if A is valid or not.

Satisfiability & Validity

A is *satisfiable* iff $v(A) = T$ for *some* interpretation.

A satisfying interpretation is called a *model* for A.

A is *valid*, denoted $\vDash A$, iff $v(A) = T$ for *all* interpretations.

A valid propositional formula is also called a *tautology*.

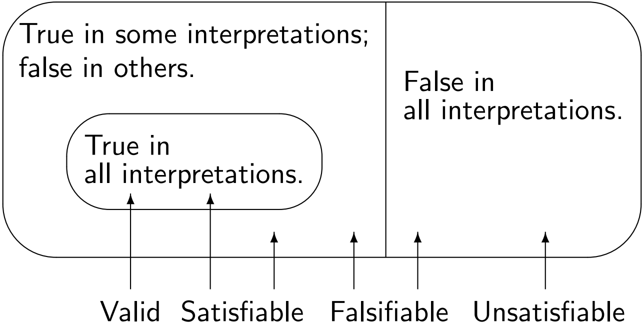
A is *unsatisfiable* iff it is not satisfiable, that is if $v(A) = F$ for *all* interpretations.

Theorem: A is valid iff $\neg A$ is unsatisfiable.

Since any formula A contains a finite number of atoms, there is a finite number of different interpretations and these can be checked in order to decide if A is valid or not.

This algorithm is called the method of truth tables since it is convenient to use a column for each atom in A and a column with the truth value for A, and a row for each interpretation.

Classification of Formulas



Validity is the key concept.

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Simple principle: Search *systematically* for a model (a satisfying interpretation).

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Simple principle: Search *systematically* for a model (a satisfying interpretation).

If a model cannot be found then the formula is unsatisfiable.

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Simple principle: Search *systematically* for a model (a satisfying interpretation).

If a model cannot be found then the formula is unsatisfiable.

For deciding validity of a formula, start by negating it (refutation by search for a falsifying counterexample).

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Simple principle: Search *systematically* for a model (a satisfying interpretation).

If a model cannot be found then the formula is unsatisfiable.

For deciding validity of a formula, start by negating it (refutation by search for a falsifying counterexample).

Soundness and completeness can be established for tableaux, implying that the method of truth tables and the method of tableaux match with respect to validity.

Tableaux

The method of truth tables is not efficient (clearly exponential) and does not work in general for FOL (arbitrary non-empty domains).

The method of semantic tableaux is a relatively efficient algorithm for deciding satisfiability (and by duality validity).

Simple principle: Search *systematically* for a model (a satisfying interpretation).

If a model cannot be found then the formula is unsatisfiable.

For deciding validity of a formula, start by negating it (refutation by search for a falsifying counterexample).

Soundness and completeness can be established for tableaux, implying that the method of truth tables and the method of tableaux match with respect to validity.

The algorithm for tableau construction is not deterministic, but it terminates with a so-called completed tableau.

Alpha-Rules for Tableaux

α	α_1	α_2
$\neg\neg A_1$	A_1	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$
$\neg(A_1 \uparrow A_2)$	A_1	A_2
$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$
$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$
$\neg(A_1 \oplus A_2)$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$

Important: α-rules create a single child.

Beta-Rules for Tableaux

β	β_1	β_2
$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$B_1 \vee B_2$	B_1	B_2
$B_1 \rightarrow B_2$	$\neg B_1$	B_2
$B_1 \uparrow B_2$	$\neg B_1$	$\neg B_2$
$\neg(B_1 \downarrow B_2)$	B_1	B_2
$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
$B_1 \oplus B_2$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$

Important: β-rules create two children.

A Closed Tableau

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

A Closed Tableau

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

Using L and R to indicate branches in the tree.

A Closed Tableau

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

Using L and R to indicate branches in the tree.

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

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

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

L

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

R

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

The tableau is closed, hence $(p \vee q) \to (q \vee p)$ is valid.

An Open Tableau

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

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

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

$$p$$

$$\odot$$

$$\text{R}$$

$$q \wedge \neg q$$

$$q, \neg q$$

$$\times$$

The tableau is open, hence $p \vee (q \wedge \neg q)$ is satisfiable.

An Open Tableau

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

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

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

$$p$$

$$\odot$$

$$\mathsf{R}$$

$$q \wedge \neg q$$

$$q, \neg q$$

$$\times$$

The tableau is open, hence $p \vee (q \wedge \neg q)$ is satisfiable.

Normally one starts with the negation of a formula in order to check for validity of the formula.

Main Rules for Tableaux

α	α_1	α_2
$\neg\neg A_1$	A_1	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$
$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$
$\neg(A_1 \oplus A_2)$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$

β	β_1	β_2
$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$B_1 \vee B_2$	B_1	B_2
$B_1 \rightarrow B_2$	$\neg B_1$	B_2
$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
$B_1 \oplus B_2$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$