Propositional Logic

Definitions

Vojtěch Bartík, March 2023

```
(* :Name: PropositionalLogic *)
(* :Title: Propositional Logic *)
(* :Author: Vojtěch Bartík *)
(* :Summary: *)
(* :Context: PropositionalLogic` *)
(* Package Version: final, March 2023 *)
(* :Mathematica Version: Mathematica 12.2 *)
(* :History: First version was created in Mathematica 2.2.1 in 1994--1995 when
   the author teached propositional logic at Faculty of Electricity of Czech
   Technical University in Prague. In 2019 and 2020 this first version was
   translated and to Mathematica 8.0 and expanded using the package "Notation"
   and graphic functions not available in version 2.2.1. The present version is
   only an adaptation to Mathematica 12.2 of the version for Mathematica 8.0.*)
```

BeginPackage

```
BeginPackage["PropositionalLogic`", {"Notation`"}];
Off[Unset::norep, Notation::noexbtag];
```

Usage

```
PropositionalLogic::usage =
 "Defines non-system logical connectives, admissible
   logical variables, logical strings representing logical formulae
   in Polish (prefix) notation, conversions between them, and many
   functions operating on logical formulae and their sets, such as a
   function computing the truth table, functions converting logical
   formulae to conjunctive and disjunctive normal forms, functions
   computing resolvents and sequences of resolvents of a set of clauses,
   a function realizing as a table simple resolution algorithms suitable
   for the manual verification of the satisfiability of a set of clauses,
   and a function realizing the algorithm described in Anthony
   Galton`s book Logic for Information Technology (John Wiley
   & Sons, 1990, pp.104-105) and finding and printing a
   refutation tree if the set of clauses is not satisfiable."
```

Logical Connectives

```
(* Logical Connectives *)
AND::usage =
  "Logical connective with obvious meaning and with infix notation x ∧ y";
EQUIV::usage =
  "Logical connective with obvious meaning and with infix notation x ⇔ y.";
IMPLIES::usage =
  "Logical connective with obvious meaning and with infix notation x ⇒ y.";
NAND::usage = "NAND[x,y] = True iff AND[x,y] = False. Infix notation x ↑ y";
NOR::usage = "NOR[x,y] = True iff OR[x,y] = False. Infix notation x ↓ y";
NOT::usage = "Logical connective with obvious meaning and with prefix notation @x";
OR::usage = "Logical connective with obvious meaning and with infix notation x ∨ y.";
XOR::usage =
  "XOR[x,y] = True iff exactly one of x and y is True. Infix notation x ⊕ y";
Connectives::usage = "List of all connectives.";
```

Logical Variables, Formulae, Strings and Conversions and their Conversions

```
(* Logical Variables, Formulae, Strings and Conversions and their Conversions *)
```

```
FALSE::usage = "Unsatisfiable formula.";
```

```
LAtomQ::usage =
  "LAtomQ[x] tests whether x is a logical variable. Admissible are letters
    of the English alphabet, symbols consisting of a single letter
    and digits, and single letters subscripted with a natural number.";
```

```
LFormulaQ::usage =
  "LFormulaQ[x] tests whether x is a logical formula. FALSE, TRUE and Hold[x],
    HoldForm[x], where x is a logical formula, are also logical formulae.";
```

```
LStringQ::usage =
  "LStringQ[x,Trace->False] tests whether x is a logical string, i.e. a
    logical formula in the Polish (prefix) notation. With the option
    Trace->True it also prints the list of steps to the result.";
```

```
LVariableSet::usage =
  "LVariableSet[x] extractes all logical
    variables from logical formula or set of logical formulae x.";
```

```
ToLFormula::usage =
  "ToLFormula[x,Trace->False] returns a corresponding logical
    formula, if x is a logical string, i.e. logical formula in the
    Polish notation, or $Failed in the opposite case. With the option
    Trace->True it also prints the list of steps to the result.";
```

```
ToLString::usage =
  "ToLString[x,Trace->False] returns a corresponding logical string, if
    x is a logical formula, or $Failed in the opposite case. With the
    option Trace->True it also prints the list of steps to the result.";
```

```
TRUE::usage = "Always true formula.";
```

Functions

```
(* Functions *)
```

```
BT::usage = "Head of vertices of a binary tree.";
BTreeQ::usage = "Defines binary tree.";
BTreeForm::usage =
  "Outputs binary tree in a convenient graphic form. It has the following
    options (default value = the first alternative):
There are the following options (default value = the first alternative):
- RootPosition -> Right|Left|Top|Bottom,
- TableSpacing -> Automatic|{nonnegative integer, nonnegative integer},
- Format -> List|TreePlot|TableForm.";
```

```
ClauseQ::usage = "Tests whether the expression is clause or not.";
```

```
CNF::usage =
  "CNF[x,type] converts the logical formula x to a minimal conjunctive
    normal form in the case type = Minimal|0 and to the complete
    conjunctive normal form in the case type = Complete|1. In the
    case type = Minimal|0 the system function BooleanConvert is used.
Attributes[CNF] = {Listable}.";
```

```
ConvertFormula::usage =
  "ConvertFormula[x, form] uses system function BooleanConvert[#,form]&
    and finds a formula y tautologically equivalent to x and
    containing only connectives determined by the argument form:
- if form is \"AND\", formula y contains only connectives AND and NOT,
- if form is \"OR\", formula y contains only connectives OR and NOT,
- if form is \"IMPLIES\", formula y contains only connectives IMPLIES and NOT,
- if form is \"NAND\", formula y contains only connectives NAND and NOT,
- if form is \"NOR\", formula y contains only connectives NOR and NOT,
- if form is \"XOR\", formula y contains only connectives XOR, AND and NOT.
Attributes[ConvertFormula] = {Listable}.";
```

```
DNF::usage =
  "DNF[x,type] converts the logical formula x to a minimal disjunctive
    normal form in the case type = Minimal|0 and to the complete
    disjunctive normal form in the case type = Complete|1. In the
    case type = Minimal|0 the system function BooleanConvert is used.
Attributes[DNF] = {Listable}.";
```

```
EmptyTruthTable::usage =
  "EmptyTruthTable[x,options] is a
    version of TruthTable with no truth-values for formulas.
There are the following options (default value = the first alternative):
- BooleanValues -> {0,1}|{False,True}|{FalseSymbol, TrueSymbol},
- ReverseValues -> False|True,
- ItemSize -> Automatic|Full|as for Grid,
- Labels -> None|Automatic|list of sufficient length,
- Print -> False|True,
- TableBreaks -> None|integer > 1|increasing list of integers > 1,
- Transpose -> False|True.
ItemSize is an option of Grid
    and Transpose is an option of List, Matrix and Tensor.";
```

```
FullResolutionSequence::usage =
  "FullResolutionSequence[x, options], where x is a list of clauses, returns a
    finite sequence x, R[1,x], R[2,x], R[3,x],..., R[n,x], where R[k,x]
    is the list of all resolvents generated from clauses in the list
    Join[x,R[1,x],...,R[k-1,x]] and satisfying certain condition determined
    by options Reduce and Rules. The last member is either empty or contains
    FALSE. In the case Rules -> True the output contains also certain rules
    that are generated and applied before each step R[k,x] in order to reduce
    the number of clauses and to find truth values for logical variables
    in case the list x is satisfiable. In general case, however, the rules
    found may not be sufficient to transform all the formulae from x to True.
There are the following options (default value = the first alternative):
- Print -> True|False,
- Reduce -> True|False,
- Rules -> True|False,
- Sort -> Union|DeleteDuplicates|False,
- Variables -> All|list of logical variables.";
```

```
HList::usage =
  "HList[x,y,z,...] and HList[{x,y,z,...}]
    return list of all arguments x,y,z,... wrapped in HoldForm.";
```

```
LEquivalentQ::usage = "LEquivalentQ[x,y] tests whether
    the logical formulae x and y are tautologically equivalent.";
LTautologyQ::usage =
  "LTautologyQ[x] tests whether the logical formula is a tautology.
Attributes[LTautologyQ] = {Listable}.";
```

```
LiteralQ::usage = "LiteralQ[x]tests whether x is a literal,
    i.e. a logical variable or negative of a logical variable.";
```

```
NormalizeLString::usage =
  "Reduces all substrings NOT@, ~AND~, ~EQUIV~, ~IMPLIES~, ~NAND~, ~NOR~,
    ~OR~, ~XOR~ to corresponding string forms of logical symbols and
    all sequences of any types of spaces in string to one RawSpace.";
```

```
ReduceSpaces::usage =
  "Reduces each sequence of any types of spaces in a string to one RawSpace.";
```

```
RefutationTree::usage =
  "RefutationTree[x,options] decides whether the list x of clauses is satisfiable or
    not. In the latter case it finds a refutation tree using the algorithm
    described in the book Logic for Information Technology (Anthony Galton, John
    Wiley & Sons, 1990, pp.104 - 105) and outputs it in a convenient graphic form.
There are the following options (default value = the first alternative):
- Format -> TreePlot|TableForm.
- RootPosition -> Right|Left|Top|Bottom.
- TableSpacing -> Automatic|{nonnegative integer, nonnegative integer}.
Option TableSpacing matters only in the case of Format -> TableForm";
```

```
ReleaseHE::usage = "ReleaseHE[x] removes all occurences
    of Hold, HoldForm and applies Evaluate on all arguments.";
```

```
Resolvents::usage =
  "Resolvents[x,options] generates from the list x of clauses the list xx
    of all resolvents with respect to the list of logical variables.
There are the following options (default value = the first alternative):
- Reduce -> True|False,
- Sort -> Union|DeleteDuplicates|False,
- Variables -> All|list of logical variables.
With Reduce → False the list xx does not include
    clauses that are members of x and with Reduce ->True it does
    not include clauses a part of which is a member of x or xx.";
```

```
ResolutionDepth::usage =
  "ResolutionDepth[x,options] characterizes the complexity of the list x of
    clauses by the triple depth={±n,r,t},where n is the length of the list
    FullResolutionSequence[x,options], r is the total number of clauses
    in it, and t is the CPU time spent in the Mathematica kernel. The sign
```

```
    is \"+\" if the list x is satisfiable,and \"-\" in the opposite case.
Alternatives for options (default value=the first alternative):
- Reduce → True|False,
- Rules → True|False,
- Sort → Union|DeleteDuplicates,
- Trace → True|False,
- Variables → All|list of logical variables.";
```

```
ResolutionSequence::usage =
  "ResolutionSequence[x, options] tests whether the list x of clauses is
    satisfiable or not and in the positive case finds values of logical
    variables for which all clauses in x are true. The algorithm used can
    be roughly described as a successive elimination of logical variables.
There are the following options (default value = the first alternative):
- Abbreviations -> {False->0, True->1}|{False->FalseSymbol, True->TrueSymbol}|None,
- Print -> True|False,
- Reduce -> True|False,
- Rules -> True|False,
- SequenceBreaks -> None|Automatic|integer > 1|incresing list of integers > 1,
- Sort -> Union|DeleteDuplicates|False,
- Variables -> Automatic|All|list of logical variables.";
```

```
ResolutionTable::usage =
  "ResolutionTable[x, options] tests whether the list x of clauses
    is satisfiable or not. It uses almost the same algorithm as
    ResolutionSequence does but the result is presented in the form of a table.
There are the following options (default value = the first alternative):
- Abbreviations -> {False->0, True->1}|{False->FalseSymbol, True->TrueSymbol}|None,
- ItemSize -> Automatic|{{Automatic, {1.5}}, 1}|as for Grid,
- Reduce -> True|False,
- Rules -> True|False,
- Sort -> Union|DeleteDuplicates|False,
- TableBreaks -> None|integer > 1|increasing list of integers > 1,
- Transpose -> False|True,
- Variables -> All|List of logical variables.";
```

```
RTree::usage =
  "RTree[x] decides whether the list x of clauses is satisfiable or not,and in the
    latter case it finds a refutation tree by the algorithm described in Anthony
    Galton:Logic for Information Technology, John Wiley& Sons,1990,pp.104-105.";
```

```
ToClauses::usage = "ToClauses[x] converts its minimal CNF
    form to the list of clauses. ToClauses has attribute Listable";
```

```
TruthTable::usage =
  "TruthTable[x,options] computes the truth table of the logical formula
    or list of logical formulae x and outputs or prints it as Grid.
There are the following options (default value = the first alternative):
```

```
- BooleanValues -> {0,1}|{False,True} -> {FalseSymbol,TrueSymbol},
- ReverseValues -> False|True,
- ItemSize -> Automatic|Full|as for Grid,
- Labels -> None|Automatic|list of sufficient length,
- Print -> False|True,
- SelectValuations ->
    All|AllTrue|Mixed|AllFalse|LastTrue|LastFalse|OnlyLastFalse|
        {{indexes|labels of formulae} -> FalseSymbol,
    {indexes|labels of formulae} -> TrueSymbol},
- TableBreaks -> None|integer > 1|increasing list of integers > 1,
- Transpose -> False|True.
ItemSize is an option of Grid
    and Transpose is an option of List, Matrix and Tensor.";
```

Function Options

```
(* Function Options *)
```

```
AllFalse::usage = "One of alternative values for SelectValuations.";
AllTrue::usage = "One of alternative values for SelectValuations.";
BooleanValues::usage = "Option of TruthTable.";
Labels::usage = "Option of TruthTable.";
LastFalse::usage = "One of alternative values for SelectValuations.";
LastTrue::usage = "One of alternative values for SelectValuations.";
Mixed::usage = "One of alternative values for SelectValuations.";
OnlyLastFalse::usage = "One of alternative values for SelectValuations.";
ReverseValues::usage = "Option of TruthTable.";
RootPosition::usage = "Option of BTreeForm and RefutationTree.";
Rules::usage =
  "Option of FullResolutionSequence, ResolutionDepth, ResolutionSequence,
    ResolutionTable and ResolutionTable.";
SelectionRule::usage = "ResolutionSequence, ResolutionTable, RTree, RefutationTree.";
SelectValuations::usage = "Option of TruthTable.";
SequenceBreaks::usage = "Option of ResolutionSequence.";
TableBreaks::usage = "Option of TruthTable and ResolutionTable.";
TestResult::usage = "Option of ResolutionTable.";
```

Notation for Logical Connectives

```
InfixNotation[ ∧ , AND];
InfixNotation[ ⇔ , EQUIV];
InfixNotation[ ⇒ , IMPLIES];
InfixNotation[ ↑ , NAND];
InfixNotation[ ↓ , NOR];
InfixNotation[ ∨ , OR];
InfixNotation[ ⊕ , XOR];
```

```
InfixNotation[ ∧ , AND, WorkingForm → TraditionalForm];
InfixNotation[ ⇔ , EQUIV, WorkingForm → TraditionalForm];
InfixNotation[ ⇒ , IMPLIES, WorkingForm → TraditionalForm];
InfixNotation[ ↑ , NAND, WorkingForm → TraditionalForm];
InfixNotation[ ↓ , NOR, WorkingForm → TraditionalForm];
InfixNotation[ ∨ , OR, WorkingForm → TraditionalForm];
InfixNotation[ ⊕ , XOR, WorkingForm → TraditionalForm];
```

```
Notation[ ¬ x_ ⇔ NOT[x_] ];
Notation[ x_ ⇔ y_ ⇔ EQUIV[x_, y_] ];
Notation[ x_ ⇒ y_ ⇔ IMPLIES[x_, y_] ];
Notation[ x_ ↑ y_ ⇔ NAND[x_, y_] ];
Notation[ ¬ x_ ↑ y_ ⇔ NAND[NOT[x_], y_] ];
Notation[ x_ ↑ ¬ y_ ⇔ NAND[x_, NOT[y_]] ];
Notation[ x_ ↓ y_ ⇔ NOR[x_, y_] ];
Notation[ ¬ x_ ↓ y_ ⇔ NOR[NOT[x_], y_] ]
Notation[ x_ ↓ ¬ y_ ⇔ NOR[x_, NOT[y_]] ];
Notation[ x_ ⊕ y_ ⇔ XOR[x_, y_] ];
Notation[ ¬ x_ ⊕ y_ ⇔ XOR[NOT[x_], y_] ]
```

```
Notation[ x_ ⊕ ¬ y_ ⟺ XOR[x_, NOT[y_]] ];
```

```
Notation[ ¬ x_ ⟺ NOT[x_] , WorkingForm → TraditionalForm];
Notation[ x_ ⟺ y_ ⟺ EQUIV[x_, y_] , WorkingForm → TraditionalForm];
Notation[ x_ ⇒ y_ ⟺ IMPLIES[x_, y_] , WorkingForm → TraditionalForm];
Notation[ x_ ↑ y_ ⟺ NAND[x_, y_] , WorkingForm → TraditionalForm];
Notation[ ¬ x_ ↑ y_ ⟺ NAND[NOT[x_], y_] , WorkingForm → TraditionalForm];
Notation[ x_ ↑ ¬ y_ ⟺ NAND[x_, NOT[y_]] , WorkingForm → TraditionalForm];
Notation[ x_ ↓ y_ ⟺ NOR[x_, y_] , WorkingForm → TraditionalForm];
Notation[ ¬ x_ ↓ y_ ⟺ NOR[NOT[x_], y_] , WorkingForm → TraditionalForm];
Notation[ x_ ↓ ¬ y_ ⟺ NOR[x_, NOT[y_]] , WorkingForm → TraditionalForm];
Notation[ x_ ⊕ y_ ⟺ XOR[x_, y_] , WorkingForm → TraditionalForm];
Notation[ ¬ x_ ⊕ y_ ⟺ XOR[NOT[x_], y_] , WorkingForm → TraditionalForm];
Notation[ x_ ⊕ ¬ y_ ⟺ XOR[x_, NOT[y_]] , WorkingForm → TraditionalForm];
```

Private

```
Begin["`Private`"];
```

Auxiliary Definitions

```
Unprotect[Union];
Clear[Union];
Union[x_Subscript] := x;
Union[x_Symbol] := x;
Protect[Union];

Unprotect[DeleteDuplicates];
Clear[DeleteDuplicates];
DeleteDuplicates[x_Subscript] := x;
DeleteDuplicates[x_Symbol] := x;
DeleteDuplicates[f_[x__]] := f @@ DeleteDuplicates[{x}] /; f =!= List;
Protect[DeleteDuplicates];

Unprotect[SubscriptBoxToString];
Clear[SubscriptBoxToString];
SubscriptBoxToString[string_] :=
  StringReplace[StringReplace[string, {"\), \(" → "$"}],
   {"\!\(\*SubscriptBox[\(" → "", "\)]\)" → ""}];
Protect[SubscriptBoxToString];
```

```
Unprotect[ToLettersAndDigits];
Clear[ToLettersAndDigits];
ToLettersAndDigits[string_] :=
  ToString /@ Map[ToExpression, Flatten[StringCases[string, #] & /@
       {WordBoundary : LetterCharacter ... ~~ LetterCharacter, "$",
        WordBoundary : DigitCharacter ... ~~ DigitCharacter}]] /;
   StringMatchQ[string, LetterCharacter ~~ ___ ~~ DigitCharacter ~~ ""];
ToLettersAndDigits[string_] := string;
Protect[ToLettersAndDigits];

Unprotect[RestoreSubscript];
Clear[RestoreSubscript];
RestoreSubscript[False] = False;
RestoreSubscript[True] = True;
RestoreSubscript[x_Symbol] := RestoreSubscript[ToString @ x];
RestoreSubscript[x_String] := x /; StringMatchQ[x, "," ~~ Whitespace];
RestoreSubscript[x_String] := ToExpression[x] /; StringFreeQ[x, "$"];
RestoreSubscript[x_String] :=
  Subscript @@ ToExpression /@ StringSplit[x, "$"] /; ! StringFreeQ[ToString[x], "$"];
Protect[RestoreSubscript];

Unprotect[RemoveSubscript];
Clear[RemoveSubscript];
RemoveSubscript[x_] := x /.
   Subscript[u_, v_] :> ToExpression[ToString /@ {u, Global`$, v} // StringJoin];
Protect[RestoreSubscript];
```

```
Unprotect[ReduceSpaces];
Clear[ReduceSpaces];
ReduceSpaces[x_String] :=
  Module[{xx = x}, xx = StringReplace[xx, {"" → " ", " " → " "}];
   FixedPoint[StringReplace[#, "  " → " "] &, xx]];
Protect[ReduceSpaces];
```

```
Unprotect[HList];
ClearAll[HList];
SetAttributes[HList, HoldAll];
HList[x_List] := Map[HoldForm, Hold[x], {2}][[1]];
HList[x__] := List @@ HoldForm /@ Hold[x];
Protect[HList];

Unprotect[ReleaseHE];
Clear[ReleaseHE];
ReleaseHE[x_] := FixedPoint[MapAll[Evaluate, #] //. (Hold | HoldForm)[u_] → u &, x];
ReleaseHE[x_, n_] := Nest[Map[Evaluate, #, n] //. (Hold | HoldForm)[u_] → u &, x, n];
Protect[ReleaseHE];
```

Basic Definitions for Logical Connectives

```
PrefixConnectives = {NOT};
```

```
InfixConnectives = {AND, EQUIV, IMPLIES, NAND, NOR, OR, XOR};
Connectives = {NOT, AND, EQUIV, IMPLIES, NAND, NOR, OR, XOR};

PrefixStrings[1] = {"¬"};
PrefixStrings[2] = {"NOT@"};
InfixStrings[1] = {"∧", "⇔", "⇒", "↑", "↓", "∨", "⊕"};
InfixStrings[2] =
  {"~AND~", "~EQUIV~", "~IMPLIES~", "~NAND~", "~NOR~", "~OR~", "~XOR~"};
Strings[1] = {"¬", "∧", "⇔", "⇒", "↑", "↓", "∨", "⊕"};
Strings[2] =
  {"NOT@", "~AND~", "~EQUIV~", "~IMPLIES~", "~NAND~", "~NOR~", "~OR~", "~XOR~"};
Strings[3] = {"NOT", "AND", "EQUIV", "IMPLIES", "NAND", "NOR", "OR", "XOR"};
Strings[4] = StringJoin[" ", #, " "] & /@ Strings[1];
```

```
Unprotect /@ Connectives;
ClearAll /@ Connectives;

AND[] = True;
AND[x_] = x;
AND[x__] := False /; MemberQ[{x}, False | FALSE];
AND[x__] := True /; Union[{x}] === {True} | {TRUE} | {True, TRUE};
AND[x___, True | TRUE, y___] = AND[x, y];

EQUIV[x_, x_] = True;
EQUIV[False | FALSE, True | TRUE] = False;
EQUIV[True | TRUE, False | FALSE] = False;
IMPLIES[False | FALSE, x_] = True;
IMPLIES[x_, True | TRUE] = True ;
IMPLIES[True | TRUE, False | FALSE] = False;
IMPLIES[True | TRUE, x] = x;

NAND[] = False;
NAND[x_] = NOT[x];
NAND[x___, False | FALSE, y___] = True;
NAND[x___, True | TRUE, y___] = NAND[x, y];
NAND[x__] := False /; AND[x] === True;
NAND[x__] := True /; AND[x] === False;

NOR[] = True;
NOR[x_] = NOT[x];
NOR[x___, True | TRUE, y___] = False; (* Corrected from True on March 21, 2023 *)
NOR[x___, False | FALSE, y___] := NOR[x, y];
NOR[x__] := False /; OR[x] === True;
NOR[x__] := True /; OR[x] === False;

NOT[False | FALSE] = True;
NOT[True | TRUE] = False;
NOT[NOT[x_]] = x;

OR[] = False;
```

```
OR[x_] = x;
OR[x___, True | TRUE, y___] = True;
OR[x___, False | FALSE, y___] = OR[x, y];

XOR[] = False;
XOR[x_] = x;
XOR[x_, x_] = False;
XOR[x___, False | FALSE, y___] = XOR[x, y];
XOR[x___, True | TRUE, y___] = NOT[XOR[x, y]];

SetAttributes[Evaluate[Connectives], HoldAll];
Protect /@ Connectives;
```

```
Unprotect[Precedence];
Clear[Precedence];
Precedence[NOT] = 230;
Precedence[AND] = 215;
Precedence[EQUIV] = 205;
Precedence[IMPLIES] = 200;
Precedence[NAND] = 215;
Precedence[NOR] = 215;
Precedence[OR] = 215;
Precedence[XOR] = 215;
Protect[Precedence];
```

Logical Variables (or Elementary Formulas or Atoms)

```
Unprotect[LAtomQ];
ClearAll[LAtomQ];
Attributes[LAtomQ] = {};
LAtomQ[x_Symbol] := True /; MemberQ[CharacterRange["a", "z"], ToString[x]];
LAtomQ[x_Symbol] := True /; MemberQ[CharacterRange["A", "Z"], ToString[x]];
LAtomQ[x_Symbol] :=
  True /; MemberQ[CharacterRange["a", "z"], StringTake[x // ToString, 1]] &&
    DigitQ[StringDrop[x // ToString, 1]];
LAtomQ[x_Symbol] := True /; MemberQ[CharacterRange["A", "Z"],
     StringTake[x // ToString, 1]] && DigitQ[StringDrop[x // ToString, 1]];
LAtomQ[x_Symbol_n_] := True /; MemberQ[CharacterRange["a", "z"], ToString[x]] &&
    n ∈ Integers && n ≥ 0;
LAtomQ[x_Symbol_n_] := True /; MemberQ[CharacterRange["A", "Z"], ToString[x]] &&
    n ∈ Integers && n ≥ 0;
LAtomQ[x_Symbol] := True /; Length[Characters[x // ToString]] > 2 &&
    MemberQ[CharacterRange["a", "z"], StringTake[x // ToString, 1]] &&
    Characters[ToString[x]][[2]] === "$" && DigitQ[StringDrop[x // ToString, 2]];
LAtomQ[x_Symbol] := True /; Length[Characters[x // ToString]] > 2 &&
    MemberQ[CharacterRange["A", "Z"], StringTake[x // ToString, 1]] &&
    Characters[ToString[x]][[2]] === "$" && DigitQ[StringDrop[x // ToString, 2]];
LAtomQ[x_] := False;
Protect[LAtomQ];
```

```
Unprotect[LVariableSet];
ClearAll[LVariableSet];
Attributes[LVariableSet] = {};
LVariableSet[x_] :=
  {x} //. Map[# → List &, {Hold, HoldForm} ~Join~ Connectives] // Flatten //
    Union // DeleteCases[#, FALSE | TRUE] & // Select[#, LAtomQ] &;
Protect[LVariableSet];
```

Logical Formulae (or Logical Expressions) and Strings

LFormulaQ[*x*] tests whether the expression *x* a logical formula. The output is True if it does and False in the opposite case. With the option Trace → True the main steps to the result are displayed.

```
Unprotect[LFormulaQ];
ClearAll[LFormulaQ];
SetAttributes[LFormulaQ, Listable];
LFormulaQ[Hold[x_]] = LFormulaQ[x];
LFormulaQ[HoldForm[x_]] = LFormulaQ[x];
LFormulaQ[PrecedenceForm[x_, y_Integer]] = LFormulaQ[x];
LFormulaQ[AND[x_, y__]] := And[LFormulaQ[x], LFormulaQ[AND[y]]];
LFormulaQ[EQUIV[x_, y_]] := And[LFormulaQ[x], LFormulaQ[y]];
LFormulaQ[IMPLIES[x_, y_]] := And[LFormulaQ[x], LFormulaQ[y]];
LFormulaQ[NAND[x_, y_]] := And[LFormulaQ[x], LFormulaQ[NAND[y]]];
LFormulaQ[NOR[x_, y__]] := And[LFormulaQ[x], LFormulaQ[NOR[y]]];
LFormulaQ[NOT[x_]] := LFormulaQ[x];
LFormulaQ[OR[x_, y__]] := And[LFormulaQ[x], LFormulaQ[OR[y]]];
LFormulaQ[XOR[x_, y__]] := And[LFormulaQ[x], LFormulaQ[XOR[y]]];
LFormulaQ[False | FALSE] = True;
LFormulaQ[True | TRUE] = True;
LFormulaQ[x_] := True /; LAtomQ[x];
LFormulaQ[x_] := False;
Protect[LFormulaQ];
```

LStringQ[*x*] tests whether the string *x* corresponds to a logical formula in the prefix (= Polish) notation. The output is True if it does and False in the opposite case. With the option Trace → True the main steps to the result are displayed.
Alternatives for options (default value = the first alternative) :
- Alignment -> Center | Left | Right,
- Trace -> False | True.

```
Unprotect[LStringQ];
ClearAll[LStringQ];
Attributes[LStringQ] = {Listable};
Options[LStringQ] = {Alignment → Center, Trace → False};

LStringQ[x_String, opts___Rule] := False /; x === "";
LStringQ[x_String, opts___Rule] :=
  False /; MemberQ[Private`Strings[1], StringTake[x, -1]];
LStringQ[x_String, opts___Rule] :=
```

```
  Module[{alignment, p, positions, trace, xx, xx1, xxx},
   {alignment, trace} = {Alignment, Trace} /. {opts} /. Options[LStringQ];
   xx = StringReplace[x, Thread[Rule[Strings[2], Strings[1]]]];
   xx = StringReplace[xx, Map[Rule[#, " " ~~ # ~~ " "] &, Strings[1]]] //
      ReduceSpaces // StringTrim;
   xxx = {{xx}};
   If[! StringFreeQ[xx, "Subscript"],
    xxx = Append[xxx, {xx = SubscriptBoxToString[xx]}]];
   xx = ReadList[StringToStream[xx], Word];
   If[xx === {}, Print[False]];
   xx1 = Select[xx, FreeQ[Strings[1], #] &];
   xx1 = RestoreSubscript /@ Select[xx1, LAtomQ[# // ToExpression] === False &];
   If[xx1 =!= {},
    Return[If[trace, StringForm[
       "False: Members of the list `` are not logical variables.", xx1], False]]];
   xx = xx /. Thread[Rule[Strings[1], Strings[3]]];
   xx = If[MemberQ[StringFreeQ[#, "$"] & /@ xx, False],
     RestoreSubscript /@ xx, ToExpression /@ xx];
   xxx = Append[xxx, xx];
   positions = Position[xx, Apply[Alternatives, Connectives]];
   positions = Flatten[positions];
   While[positions =!= {},
    p = Last[positions];
    Which[And[NOT === Part[xx, p], p ≤ Length[xx] - 1],
     xx = Drop[ReplacePart[xx, p → True], {p + 1}]; AppendTo[xxx, xx],
     And[NOT === Part[xx, p], p == Length[xx]], xxx,
     And[NOT =!= Part[xx, p], p + 2 ≤ Length[xx]],
     xx = Drop[ReplacePart[xx, p → True], {p + 1, p + 2}];
     AppendTo[xxx, xx],
     And[NOT =!= Part[xx, p], p + 2 > Length[xx]], xxx];
    positions = Drop[positions, -1]];
   If [Length[Last@xxx] > 1, AppendTo[xxx, {False}]];
   If[trace, xxx // Column[#, alignment] &, Last@Last@xxx]];
Protect[LStringQ];
```

```
Clear[NormalizeLString];
NormalizeLString[s_String] := FixedPoint[ReduceSpaces,
    StringReplace[StringReplace[s, Thread[Rule[Strings[2], Strings[1]]]],
     Thread[Rule[Strings[1], Strings[4]]]]] // StringTrim;
```

Conversion between Logical Strings and Formulas

ToLFormula[*x*] tests whether the string *x* corresponds to a logical formula in the prefix (= Polish) notation, and returns a logical formula if the test is succesful or $Failed in the opposite case. With the option Trace → True the main steps to the result are displayed.
Alternatives for options (default value = the first alternative) :
- Alignment -> Center | Left | Right,
- Trace -> False | True.

```
Unprotect[ToLFormula];
```

```
ClearAll[ToLFormula];
Attributes[ToLFormula] = {Listable};
Options[ToLFormula] = {Alignment → Center, Trace → False};

ToLFormula[x_String, opts___Rule] := $Failed /; x === "";
ToLFormula[x_String, opts___Rule] :=
  $Failed /; MemberQ[Strings[1], StringTake[x, -1]];
ToLFormula[x_String, opts___Rule] :=
  Module[{alignment, p, positions, rules, trace, xx, xx1, xxx},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   {alignment, trace} = {Alignment, Trace} /. {opts} /. Options[ToLFormula];
   xx = StringReplace[x, Thread[Rule[Strings[2], Strings[1]]]];
   xx = StringReplace[xx, Map[Rule[#, " " ~~ # ~~ " "] &, Strings[1]]] //
       ReduceSpaces // StringTrim;;
   xxx = {{xx}};
   xx = StringReplace[xx // ReduceSpaces, {"\)", \(" → "\),\("}];
   If[! FreeQ[xx, "Subscript"], xx = SubscriptBoxToString[xx]];
   xx = ReadList[StringToStream[xx], Word];
   If[xx === {}, SetAttributes[#, HoldAll] & /@ Connectives;
    Return[False]];
   xx1 = Select[xx, FreeQ[Strings[1], #] &];
   xx1 = Select[xx1, LAtomQ[StringReplace[#, "$" → ""] // ToExpression] === False &];
   xx1 = RestoreSubscript /@ xx1;
   If[xx1 =!= {}, SetAttributes[#, HoldAll] & /@ Connectives;
    Return[If[trace, StringForm[
       "False: Members of the list `` are not logical variables.", xx1], $Failed]]];
   xx = xx /. Thread[Rule[Strings[1], Strings[3]]];
   xxx = Append[xxx, xx = Map[RestoreSubscript, xx]];
   positions = Position[xx, Apply[Alternatives, Connectives]];
   positions = Flatten[positions];
   While[positions =!= {}, p = Last[positions];
    Which[And[NOT === Part[xx, p], p + 1 ≤ Length[xx]],
     xx = Drop[ReplacePart[xx, p → PrecedenceForm[xx[[p]] @ xx[[p + 1]], 600]], {p + 1}];
     xxx = Append[xxx, xx],
     And[NOT === Part[xx, p], p == Length[xx]], xxx,
     And[NOT =!= Part[xx, p], p + 2 ≤ Length[xx],
      xx[[p + 1 ;; p + 2]] ⋂ Connectives === {}], xx = Drop[ReplacePart[xx,
        p → PrecedenceForm[xx[[p]] @@ xx[[p + 1 ;; p + 2]], 100]], {p + 1, p + 2}];
     xxx = AppendTo[xxx, xx],
     And[NOT =!= Part[xx, p], p + 2 > Length[xx]], xxx];
    positions = Drop[positions, -1]];
   SetAttributes[#, HoldAll] & /@ Connectives;
   If [Length[Last@xxx] > 1, AppendTo[xxx, {$Failed}]];
   If[trace, xxx // Column[#, alignment] &, Last@Last@xxx]];
Protect[ToLFormula];
```

ToLString[*x*] converts a logical formula *x* to a string corresponding to its prefix (Polish) version.
With the option Trace → True the main steps to the result are displayed.
Alternatives for options (default value = the first alternative) :
- Alignment -> Center | Left | Right,

- Trace -> False | True.

```
Unprotect[ToLString];
ClearAll[ToLString];
Attributes[ToLString] = {Listable};
Options[ToLString] = {Alignment → Center, Trace → False};

ToLString[x_, opts___Rule] := $Failed /; Not[LFormulaQ[x]];
ToLString[x_, opts___Rule] := ToString[x, StandardForm] /; LAtomQ[x];
ToLString[x_, opts___Rule] :=
  Module[{alignment, rules1, rules2,
    ssVariables1, ssVariables2, ss1, ss2, ss3, trace, xx, xxx},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   {alignment, trace} = {Alignment, Trace} /. {opts} /. Options[ToLString];
   xx = FixedPoint[ReleaseHold, x] /. {FALSE → False, TRUE → True};
   xxx = {{xx}};
   ssVariables1 = Cases[xx, Subscript[_, _], ∞] // Union;
   If[ssVariables1 =!= {},
    ss1 = ToString[RemoveSubscript[#]] & /@ ssVariables1;
    ssVariables2 = ToExpression /@ Map[StringJoin[#, "•"] &, ss1];
    ss2 = ToString /@ ssVariables2;
    rules1 = Thread[Rule[ssVariables1, ssVariables2]];
    ss3 = StringReplace[#, u : LetterCharacter ~~ "$" ~~ v : DigitCharacter .. ~~ "•" :>
          "\!\(\*SubscriptBox[\(" ~~ u ~~ "\), \(" ~~ v ~~ "\)]\)"] & /@ ss2;
    rules2 = Thread[Rule[ss2, ss3]];
    xxx = Append[xxx, xx = xx /. rules1; {xx}]];
   xxx = Append[xxx, xx = xx // ToString; {xx}];
   xx = StringReplace[
     StringReplace[xx, {"]" ... ~~ "]" → "]", ", " → " "}], {"[" → " ", "]" → " "}];
   xxx = Append[xxx, xx = xx // ReduceSpaces // StringTrim; {xx}];
   xxx = Append[xxx, xx = StringSplit[xx] /. Thread[Rule[Strings[3], Strings[1]]]];
   xx = Map[StringJoin[" ", #] &, xx];
   xxx = Append[xxx,
     xx = StringReplace[xx // StringJoin // ReduceSpaces // StringTrim, " " → ""];
     {xx}];
   If[ssVariables1 =!= {}, xxx = Append[xxx, xx = StringReplace[xx, rules2]; {xx}]];
   SetAttributes[#, HoldAll] & /@ Connectives;
   If[trace, xxx // Column[#, alignment] &, xx]];
Protect[ToLString];
```

TruthTable

TruthTable[x,options] computes the truth table of the logical formula or list of logical formulae x and outputs or prints it as Grid.- Alternatives for options (default value = the first alternative) :
- BooleanValues -> { 0, 1} | {False,True} | -> {FalseSymbol, TrueSymbol},
- ReverseValues -> False | True,
- ItemSize -> Automatic | Full | as for Grid,
- Labels -> None | Automatic | list of sufficient length,
- Print -> False | True,
- SelectValuations -> All | AllTrue | Mixed | AllFalse | LastTrue | LastFalse |

OnlyLastFalse | {{ (indexes | labels of) formulae} -> FalseSymbol, {(indexes | labels of) formulae} -> TrueSymbol},
- TableBreaks -> None | integer > 1 | increasing list of integers > 1,
- Transpose -> False | True,
- Variables -> All | List of logical variables.
ItemSize is an option of Grid and Transpose is an option of List, Matrix and Tensor.

```
Unprotect[TruthTable];
ClearAll[TruthTable];
Options[TruthTable] = {BooleanValues → {0, 1}, ItemSize → Automatic,
    Labels → None, Print → False, ReverseValues → False, SelectValuations → All,
    TableBreaks → None, Transpose → False, Variables → All};

TruthTable[x_List /; MemberQ[Map[LFormulaQ, x], False], options___Rule] :=
  "Failed, a member of the first argument isn't logical formula";
TruthTable[x_List, options___Rule] :=
  Module[{booleanValues, dividers, heading, itemSize, labels, position0, position1,
    print, reverseValues, selectValuations, table, tableBreaks, tableLength,
    tableParts, tableWidth, transpose, variables, variablesLength, xx},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   {booleanValues, itemSize, labels, print,
     reverseValues, selectValuations, tableBreaks, transpose, variables} =
    {BooleanValues, ItemSize, Labels, Print, ReverseValues,
       SelectValuations, TableBreaks, Transpose, Variables} /.
      Flatten[{options}] /. Options[TruthTable];
   xx = FixedPoint[ReleaseHold, x] /. {FALSE → False, TRUE → True};
   variables =
    If[variables === All, LVariableSet[xx], Select[variables, ! FreeQ[xx, #] &]];

   (* Evaluating formulae *)
   If[variables =!= {}, variablesLength = variables // Length,
    SetAttributes[#, HoldAll] & /@ Connectives;
    Return["There are no variables"]];
   table = Tuples[If[! reverseValues, {False, True}, {True, False}], variablesLength];
   table = MapThread[Join,
     {table, xx /. MapThread[Rule, #] & /@ Tuples[{{variables}, table}]}];

   (* Selecting Valuations according to the option SelectValuations *)
   Which[labels === Automatic, labels = Range[1, Length[xx]],
    Head[labels] === List && Length[labels] ≥ Length[xx],
    labels = Take[labels, Length[xx]],
    Head[labels] === List && 0 < Length[labels] < Length[xx],
    labels = Join[labels, Table[" ", {Length[xx] - Length[labels]}]],
    True, labels = False];
   Which[Head[labels] === List && Head[selectValuations] === List,
    selectValuations = selectValuations /. MapIndexed[Rule[#1, #2 // First] &, labels];
    selectValuations = selectValuations /.
      Rule[u_, v_] :↦ Rule[u, v /. Thread[Rule[booleanValues, {False, True}]]],
    labels === False && Head[selectValuations] === List,
    selectValuations = selectValuations /.
```

```
    Rule[u_, v_] :> Rule[u, v /. Thread[Rule[booleanValues, {False, True}]]]];
If[Head[selectValuations] === List, selectValuations =
  Sort[selectValuations, Order[#1[[2]], #2[[2]]] == 1 &]];

(* Selecting valuations *)
Which[selectValuations === AllTrue,
 table = Select[table, FreeQ[Drop[#, variablesLength], False] &],
 selectValuations === Mixed, table = Select[table,
   Union[Drop[#, variablesLength]] === Union[{False, True}] &],
 selectValuations === AllFalse, table = Select[table,
   FreeQ[Drop[#, variablesLength], True] &],
 selectValuations === LastTrue, table = Select[table, Last[#] === True &],
 selectValuations === LastFalse, table = Select[table, Last[#] === False &],
 selectValuations === OnlyLastFalse, table =
  Select[table, Position[Drop[#, variablesLength], False] === {{Length[xx]}} &],
 MatchQ[selectValuations, {{__Integer} → False, {__Integer} → True} |
   {{__Integer} → False} | {{__Integer} → True}],
 {position0, position1} = {False, True} /. Map[Reverse, selectValuations] /.
   {False → {}, True → {}};

 If[position0 =!= False, position0 = position0 + variablesLength // Sort;
  table = Select[table,
    (Intersection[Flatten[Position[#, False]], position0] === position0) &]];
 If[position1 =!= True, position1 = position1 + variablesLength // Sort;
  table = Select[table,
    (Intersection[Flatten[Position[#, True]], position1] === position1) &]],
 True, table];
If[table === {}, SetAttributes[#, HoldAll] & /@ Connectives;
 Return["No valuations have been selected"]];
table = table /. RuleDelayed[u_ /; FreeQ[u, Alternatives @@ Connectives],
   u /. Thread[Rule[{False, True}, booleanValues]]];
SetAttributes[#, HoldAll] & /@ Connectives;

(* Formating output *)
{tableLength, tableWidth} = Dimensions[table];
heading = Join[variables, x];
Which[tableBreaks ~ MatchQ ~ _Integer,
 tableBreaks = Range[tableBreaks + 1, tableLength, tableBreaks],
 tableBreaks ~ MatchQ ~ {__Integer}, tableBreaks =
  Select[tableBreaks + 1, # ≤ tableLength &],
 True, tableBreaks = {}];
tableBreaks = Union[{1}, tableBreaks, {tableLength + 1}];
tableParts =
 {tableBreaks[[#]], tableBreaks[[# + 1]] - 1} & /@ Range[Length[tableBreaks] - 1];
tableParts = Take[table, #] & /@ tableParts;
tableParts = Prepend[#, heading] & /@ tableParts;
If[labels =!= False,
 tableParts =
  Prepend[#, Join[Table[Global`•, {variablesLength}], labels]] & /@ tableParts;
 dividers = {{{Thick, {True}, Thick}, variablesLength + 1 → Thick},
```

```
      {{Thick, True, Thick, {True}, Thick}, -1 → Thick}},
    dividers = {{{Thick, {True}, Thick}, variablesLength + 1 → Thick},
      {{Thick, Thick, {True}, Thick}, -1 → Thick}}];
   If[transpose, tableParts = Transpose /@ tableParts;
    dividers = Reverse[dividers]];
   table = Grid[#, Dividers → dividers, ItemSize → itemSize] & /@ tableParts;
   If[print === False,
    If[Length@table == 1, First@table, table], StylePrint /@ table;]];
TruthTable[x_, options___Rule] := TruthTable[{x}, options];
Protect[TruthTable];
```

EmptyTruthTable

EmptyTruthTable[x, options] is a version of TruthTable with no truth-values for formulas.
- Alternatives for options (default value = the first alternative) :
- BooleanValues -> { 0, 1} | {False,True} | -> {FalseSymbol, TrueSymbol},
- ReverseValues -> False | True,
- ItemSize -> Automatic | Full | as for Grid,
- Labels -> None | Automatic | list of sufficient length,
- Print -> False | True,
- TableBreaks -> None | integer > 1 | increasing list of integers > 1,
- Transpose -> False | True.

```
Unprotect[EmptyTruthTable];
ClearAll[EmptyTruthTable];
Options[EmptyTruthTable] = {BooleanValues → {0, 1},
   ItemSize → Automatic, Labels → None, Print → False, ReverseValues → False,
   SelectValuations → All, TableBreaks → None, Transpose → False, Variables → All};

EmptyTruthTable[x_List /; MemberQ[Map[LFormulaQ, x], False], options___Rule] :=
  "Failed, a member of the first argument isn't logical formula";
EmptyTruthTable[x_List, options___Rule] :=
  Module[{booleanValues, dividers, heading, itemSize,
    labels, print, reverseValues, table, tableBreaks, tableLength,
    tableParts, tableWidth, transpose, variables, variablesLength, xx},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   {booleanValues, itemSize, labels, print, reverseValues, tableBreaks, transpose} =
    {BooleanValues, ItemSize, Labels, Print, ReverseValues, TableBreaks, Transpose} /.
     Flatten[{options}] /. Options[TruthTable];
   variables = LVariableSet[x];
   xx = FixedPoint[ReleaseHold, x] /. {FALSE → False, TRUE → True};
   If[variables =!= {}, variablesLength = variables // Length,
    SetAttributes[#, HoldAll] & /@ Connectives;
    Return["There are no variables"]];
   table = Tuples[If[! reverseValues, {False, True}, {True, False}], variablesLength];
   table = table /. Thread[Rule[{False, True}, booleanValues]];
   table = Map[Join[#, Table[" ", {xx // Length}]] &, table];
   Which[labels === Automatic, labels = Range[1, Length[xx]],
    Head[labels] === List && Length[labels] ≥ Length[xx],
    labels = Take[labels, Length[xx]],
```

```
    Head[labels] === List && 0 < Length[labels] < Length[xx],
    labels = Join[labels, Table[" ", {Length[xx] - Length[labels]}]],
    True, labels = False];
   SetAttributes[#, HoldAll] & /@ Connectives;
   (* Formating output *)
   {tableLength, tableWidth} = Dimensions[table];
   heading = Join[variables, x];
   Which[tableBreaks ~ MatchQ ~ _Integer,
    tableBreaks = Range[tableBreaks + 1, tableLength, tableBreaks], tableBreaks ~ MatchQ ~
     {__Integer}, tableBreaks = Select[tableBreaks + 1, # ≤ tableLength &],
    True, tableBreaks = {}];
   tableBreaks = Union[{1}, tableBreaks, {tableLength + 1}];
   tableParts =
    {tableBreaks[[#]], tableBreaks[[# + 1]] - 1} & /@ Range[Length[tableBreaks] - 1];
   tableParts = Take[table, #] & /@ tableParts;
   tableParts = Prepend[#, heading] & /@ tableParts;
   If[labels =!= False,
    tableParts =
     Prepend[#, Join[Table[Global`•, {variablesLength}], labels]] & /@ tableParts;
    dividers = {{{Thick, {True}, Thick}, variablesLength + 1 → Thick},
      {{Thick, True, Thick, {True}, Thick}, -1 → Thick}},
    dividers = {{{Thick, {True}, Thick}, variablesLength + 1 → Thick},
      {{Thick, Thick, {True}, Thick}, -1 → Thick}}];
   If[transpose, tableParts = Transpose /@ tableParts;
    dividers = Reverse[dividers]];
   table = Grid[#, Dividers → dividers, ItemSize → itemSize] & /@ tableParts;
   If[print === False,
    If[Length@table == 1, First@table, table], StylePrint /@ table;]];
EmptyTruthTable[x_, options___Rule] := EmptyTruthTable[{x}, options];
Protect[EmptyTruthTable];
```

Tests of Tautology and Tautological Equivalence

LTautologyQ[*x*] returns True if *x* is a tautology, and False if *x* is a logical formula but not a tautology.

```
Unprotect[LTautologyQ];
ClearAll[LTautologyQ];
SetAttributes[LTautologyQ, Listable];
LTautologyQ[x_] :=
  Module[{tuples, variables, xx},
    ClearAttributes[#, HoldAll] & /@ Connectives;
    xx = FixedPoint[ReleaseHold, x];
    If[xx =!= True,
     xx = xx /. FALSE → False /. TRUE → True;
     variables = LVariableSet[xx];
     tuples = Tuples[{False, True}, Length[variables]];
     xx = xx /. MapThread[Rule, #] & /@ Tuples[{{variables}, tuples}];
     SetAttributes[#, HoldAll] & /@ Connectives;
     xx = MatchQ[Union[xx], {True}]];
```

```
    xx] /; LFormulaQ[x] === True;
LTautologyQ[x_] := "Failed, the argument is not a logical formula";
Protect[LTautologyQ];
```

LEquivalentQ[*x*, *y*] returns True if *x*, *y* are logically equivalent logical formulas, and False if *x*, *y* are logically inequvalent logical formulas.

```
Unprotect[LEquivalentQ];
ClearAll[LEquivalentQ];
LEquivalentQ[x_, y_] := LTautologyQ[x ⇔ y] /; LFormulaQ[x ⇔ y] === True;
LEquivalentQ[x_, y_] := "Failed, some argument is not a logical formula";
Protect[LEquivalentQ];
```

Conjunctive and Disjunctive Normal Forms - Complete and Simplified

CNF[x, Complete | 1] returns the complete CNF of a logical formula x obtained from the truth table of x.
CNF[x, Minimal | 0] returns a simplified DNF of x obtained by the system function BooleanMinimize.
The second argument is optional and its default value is Simplified | 0.

```
Unprotect[CNF];
ClearAll[CNF];
SetAttributes[CNF, Listable];
CNF[x_ /; Not@LFormulaQ[x], t_ : Minimal] :=
  "Failed, the argument is not a logical formula";
CNF[x_, t_ : Minimal] :=
  Module[{rules, table, type, variables, varsnumber, xx = x},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   SetAttributes[{AND, OR}, {Flat, Orderless}];
   xx = FixedPoint[ReleaseHold, xx] /. {FALSE → False, TRUE → True};
   type = t /. {Complete → 1, Minimal → 0};
   Which[type == 0,
    rules =
     Thread[Rule[Connectives, {Not, And, Equivalent, Implies, Nand, Nor, Or, Xor}]];
    xx = xx //. rules // BooleanConvert[#, "CNF"] &;
    rules = Reverse /@ rules;
    xx = xx /. rules,
    type == 1,
    variables = LVariableSet[xx]; varsnumber = Length[variables];
    table = Tuples[{False, True}, varsnumber];
    rules = Map[MapThread[Rule, {variables, #}] &, table];
    table = MapThread[Join, {table, Map[({xx} /. #) &, rules]}];
    table = Map[Drop[#, -1] &, Cases[table, {___, False}]];
    table = Map[MapThread[List, {#, variables}] &, table] //.
      {{False, y_} → y, {True, y_} → NOT[y]};
    xx = Apply[AND, Map[Apply[OR, #] &, table]]];
   ClearAttributes[{AND, OR}, {Flat, Orderless}];
   SetAttributes[#, HoldAll] & /@ Connectives;
   xx];
```

```
Protect[CNF];
```

DNF[x, Complete | 1] returns the complete DNF of a logical formula x obtained from the truth table of x.DNF[x, Minimal | 0] returns a simplified DNF of x obtained by the system function BooleanMinimize.The second argument is optional and its default value is Simplified | 0.

```
Unprotect[DNF];
ClearAll[DNF];
SetAttributes[DNF, Listable];
DNF[x_ /; Not@LFormulaQ[x], t_ : Minimal] :=
  "Failed, the argument is not a logical formula";
DNF[x_, t_ : Minimal] :=
  Module[{rules, table, type, variables, varsnumber, xx = x},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   SetAttributes[{AND, OR}, {Flat, Orderless}];
   xx = FixedPoint[ReleaseHold, xx] /. {FALSE → False, TRUE → True};
   type = t /. {Complete → 1, Minimal → 0};
   Which[type == 0,
    rules =
     Thread[Rule[Connectives, {Not, And, Equivalent, Implies, Nand, Nor, Or, Xor}]];
    xx = xx //. rules // BooleanConvert[#, "DNF"] &;
    rules = Reverse /@ rules;
    xx = xx /. rules,
    type == 1,
    variables = LVariableSet[xx]; varsnumber = Length[variables];
    table = Tuples[{False, True}, varsnumber];
    rules = Map[MapThread[Rule, {variables, #}] &, table];
    table = MapThread[Join, {table, Map[({xx} /. #) &, rules]}];
    table = Map[Drop[#, -1] &, Cases[table, {___, True}]];
    table = Map[MapThread[List, {#, variables}] &, table] //.
      {{False, y_} → NOT[y], {True, y_} → y};
    xx = Apply[OR, Map[Apply[AND, #] &, table]]];
   ClearAttributes[{AND, OR}, {Flat, Orderless}];
   SetAttributes[#, HoldAll] & /@ Connectives;
   xx];
Protect[DNF];
```

Conversion to Complete Sets of Connectives: {¬ , ∧}, {¬ , ∨}, {¬ , ⇒}, {¬ , ∧ , ⊕}, {↑}, {↓}

ConvertFormula[*x*, form] uses system function BooleanConvert and finds a formula *y* tautologically equivalent to *x* and containing only connectives determined by the argument form:
- if form is "AND", formula y contains only connectives AND and NOT;
- if form is "OR", formula y contains only connectives OR and NOT;
- if form is "IMPLIES", formula y contains only connectives IMPLIES and NOT;
- if form is "NAND", formula y contains only connectives NAND and NOT;
- if form is "NOR", formula y contains only connectives NOR and NOT;
- if form is "XOR", formula y contains only connectives XOR, AND and NOT.

```
Unprotect[ConvertFormula];
Clear[ConvertFormula];
```

```
SetAttributes[ConvertFormula, Listable];
ConvertFormula[x_ /; LFormulaQ[x], form_] :=
  Module[{rules, xx},
   ClearAttributes[#, HoldAll] & /@ Connectives;
   SetAttributes[{AND, OR}, {Flat, Orderless}];
   xx = FixedPoint[ReleaseHold, x] /. {FALSE → False, TRUE → True};
   rules =
    Thread[Rule[Connectives, {Not, And, Equivalent, Implies, Nand, Nor, Or, Xor}]];
   xx = xx /. rules;
   Which[form === "AND", xx = BooleanConvert[xx, "AND"],
    form === "OR", xx = BooleanConvert[xx, "OR"],
    form === "IMPLIES", xx = BooleanConvert[xx, "IMPLIES"],
    form === "NAND", xx = BooleanConvert[xx, "NAND"],
    form === "NOR", xx = BooleanConvert[xx, "NOR"],
    form === "XOR", xx = BooleanConvert[xx, "ESOP"]];
   rules = Reverse /@ rules;
   xx = xx /. rules;
   ClearAttributes[{AND, OR}, {Flat, Orderless}];
   SetAttributes[#, HoldAll] & /@ Connectives;
   xx];
ConvertFormula[x_ /; Not[LFormulaQ], form_] :=
  "Failed, the argument is not a logical formula";
Protect[ConvertFormula];
```

Preliminaries on Clauses

```
Unprotect[LiteralQ];
Clear[LiteralQ];
LiteralQ[x_] := True /; LAtomQ[x];
LiteralQ[NOT[x_]] := True /; LAtomQ[x];
LiteralQ[x_] = False;
Protect[LiteralQ];

Unprotect[ClauseQ];
Clear[ClauseQ];
ClauseQ[] = True;
ClauseQ[OR[x___], False | FALSE | True | TRUE, OR[y___]] = ClauseQ[OR[x, y]];
ClauseQ[OR[x_, y__]] := ClauseQ[x] && ClauseQ[OR[y]];
ClauseQ[x_] := True /; LiteralQ[x];
ClauseQ[x_] := False;
Protect[ClauseQ];

Unprotect[ToClauses];
ClearAll[ToClauses];
Attributes[ToClauses] = {Listable};
ToClauses[x_ /; LFormulaQ[x]] := CNF[{x}] /. {AND → List, True → TRUE} // Flatten;
Protect[ToClauses];

Unprotect[variableCase];
Clear[variableCase];
```

```
variableCase[x_List, y_] := 0 /; MemberQ[x, y] && FreeQ[x, NOT[y], 1];
variableCase[x_List, y_] := 0 /; MemberQ[x, NOT[y]] && FreeQ[x, y, 1];
variableCase[x_List, y_] := 1 /; FreeQ[x, NOT[y]] && MemberQ[x, y, {2}];
variableCase[x_List, y_] :=
  1 /; FreeQ[x /. NOT[y] → False, y] && MemberQ[x, NOT[y], {2}];
variableCase[x_List, y_] := 2 /; MemberQ[x, y] && MemberQ[x, NOT[y]];
variableCase[x_List, y_] := 2 /; MemberQ[x, y, {2}] && MemberQ[x, NOT[y], {2}];
Protect[variableCase];

Unprotect[variableRule];
Clear[variableRule];
variableRule[x_List, y_] := {y → True} /; MemberQ[x, y] && FreeQ[x, NOT[y], 1];
variableRule[x_List, y_] := {y → False} /; MemberQ[x, NOT[y]] && FreeQ[x, y, 1];
variableRule[x_List, y_] := {y → True} /; FreeQ[x, NOT[y]] && MemberQ[x, y, {2}];
variableRule[x_List, y_] :=
  {y → False} /; FreeQ[x /. NOT[y] → False, y] && MemberQ[x, NOT[y], {2}];
variableRule[x_OR, ∞] := {∞ → x};
variableRule[x_OR, y_] := {y → True} /; MemberQ[x, y];
variableRule[x_OR, y_] := {y → False} /; MemberQ[x, NOT[y]];
variableRule[x_OR, y_] := {y → "•"} /; FreeQ[x, y] && y =!= ∞;
variableRule[NOT[x_], ∞] := {∞ → x};
variableRule[NOT[x_], y_] := {y → False} /; x === y;
variableRule[NOT[x_], y_] := {y → "•"} /; y =!= ∞;
variableRule[Global`■, y_] = {y → y};
variableRule[∞, y_] := {y → y};
variableRule[False, ∞] = {∞ → FALSE};
variableRule[x_Symbol, ∞] = {∞ → x};
variableRule[x_Symbol, y_] := {y → True} /; x === y;
variableRule[x_Symbol, y_] := {y → "•"} /; x =!= y;
variableRule[x_, y_] := {y → "⋯"} /; MemberQ[x, Global`•, 2];
Protect[variableRule];
```

Resolvents and Resolution Sequences

generateResolvents

```
Unprotect[generateResolvents];
Clear[generateResolvents];
generateResolvents[x1_, x2_, x3_, reduce_, sort_, variables_] :=
  Module[{v, variables1, x10, x11, x12, x20, x21, x22, xx = {}, xx12, xx21, xx22},
   If[x2 =!= {},
    {x10, x20} = Map[sort, {x1, x2}];
    variables1 = Select[variables, MemberQ[x20, #, -1] &];
    While[And[variables1 =!= {}, FreeQ[x20, False]],
     v = First[variables1];
     x11 = Select[x10, SameQ[# /. v → True, True] &];
     x12 = Select[x10, SameQ[# /. v → False, True] &];
     x21 = Select[x20, SameQ[# /. v → True, True] &];
     x22 = Select[x20, SameQ[# /. v → False, True] &];
     xx12 = Distribute[OR[x11 /. v → False, x22 /. NOT[v] → False], List];
     xx12 = DeleteCases[xx12, OR[u__, NOT[u__], ___]] //. OR[u_, u_] → u;
```

```
      xx21 = Distribute[OR[x21 /. v → False, x12 /. NOT[v] → False], List];
      xx21 = DeleteCases[xx21, OR[u__, NOT[u__], ___]] //. OR[u_, u_] → u;
      xx22 = Distribute[OR[x21 /. v → False, x22 /. NOT[v] → False], List];
      xx22 = DeleteCases[xx22, OR[u__, NOT[u__], ___]] //. OR[u_, u_] → u;
      xx = Join[xx, xx12, xx21, xx22];
      variables1 = Drop[variables1, 1]];
     xx = xx // sort;
     If[reduce,
      xx = DeleteCases[xx, OR[u_, __] | u_ /; MemberQ[x3, u]];
      xx = DeleteCases[xx, OR[u_, __] /; MemberQ[xx, u]],
      xx = DeleteCases[xx, u_ /; MemberQ[x3, u]]]];
    xx];
Protect[generateResolvents];
```

Resolvents[x, options] generates from the list x of clauses the list xx of all resolvents with respect to the list of logical variables given by one of the optional arguments, from which, however, some resolvents are deleted: with the option Reduce ->False, there are excluded resolvents that are members of x, and with the option Reduce ->True there are excluded resolvents a part of which is a member of x or xx.
Alternatives for options (default value = the first alternative) :
- Reduce -> True | False
- Sort -> DeleteDuplicates | Union | False,
- Variables -> All | List of logical variables.

```
Unprotect[Resolvents];
ClearAll[Resolvents];
Options[Resolvents] = {Reduce → True, Sort → DeleteDuplicates, Variables → All};

Resolvents[x_, options___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
Resolvents[x_List, options___Rule] :=
 "Failed, a member of the first argument is not clause" /;
  MemberQ[Map[ClauseQ, x], False];
Resolvents[x_, options___Rule] :=
 Module[{reduce, rules, sort, u, variables, variables1, x1, x2, x3, xx},
  ClearAttributes[{NOT, OR}, HoldAll];
  SetAttributes[OR, {Flat, Orderless}];
  {reduce, sort, variables} = {Reduce, Sort, Variables} /. {options} /.
     Options[Resolvents] /. All → LVariableSet[x];
  xx = DeleteCases[x, OR[u__, NOT[u__], ___]];
  If[xx === {},
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
   Return["Failed, all members of the first argument are tautologies"]];
  variables =
   If[variables === All, LVariableSet[xx], Select[variables, ! FreeQ[xx, #] &]];
  If[variables === {},
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
   Return["Failed, the first argument does not depend on given variables"]];
```

```
  variables1 = RemoveSubscript[variables];
  variables1 =
   ToExpression@StringReplace[ToString[#], "Private`$" → "$"] & /@ variables1;
  rules = Thread[Rule[variables, variables1]];
  xx = xx /. rules;
  xx = DeleteCases[xx, OR[u__, NOT[u__], ___]] //. OR[u_, u_] → u;
  {x1, x2, x3} = {{}, xx, xx};
  xx = generateResolvents[x1, x2, x3, reduce, sort, variables1];
  xx = xx /. Reverse /@ rules;
  ClearAttributes[OR, {Flat, Orderless}];
  SetAttributes[{NOT, OR}, HoldAll];
  xx /. False → FALSE];
Protect[Resolvents];
```

FullResolutionSequence[x, options], where x is a list of clauses and one of the optional arguments is Rules -> False, returns a list {x, Complement[R[x], x], Complement[R[R[x]], R[x],x], . . .} where R[xx] = Resolvents[xx, opts]. The last member is either empty or contains FALSE. In the case Rules -> True the output contains also certain rules that are applied repeatedly before each application of Resolvents[#, opts] & in order to reduce the number of clauses and to find truth values for logical variables in case the list x is satisfiable. In general case, however, the rules found may not be sufficient to transform all the formulae from x to True.
Alternatives for options (default value = the first alternative) :
- BooleanValues -> { 0, 1} | {False,True} | -> {FalseSymbol, TrueSymbol},
- Print -> False | True,
- Reduce -> True | False,
- Rules -> True | False,
- Sort -> DeleteDuplicates | Union | False,
- Variables -> All | List of logical variables.

```
Unprotect[FullResolutionSequence];
ClearAll[FullResolutionSequence];
Attributes[FullResolutionSequence] = {};
Options[FullResolutionSequence] = {BooleanValues → {0, 1}, Print → False,
    Reduce → True, Rules → True, Sort → DeleteDuplicates, Variables → All};

FullResolutionSequence[x_, options___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
FullResolutionSequence[x_List, options___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
FullResolutionSequence[x_, options___Rule] :=
  Module[{booleanValues, case, print, reduce, rules, rules1, rules2,
    sequence = {}, sort, variables, variables1, x1, x2, x3, xx},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
   {booleanValues, print, reduce, rules, sort, variables} =
    {BooleanValues, Print, Reduce, Rules, Sort, Variables} /. {options} /.
     Options[FullResolutionSequence];
   If[sort === False, sort = Identity];
   xx = DeleteCases[x, OR[u__, NOT[u__], ___]] //. OR[u_, u_] → u;
```

```
    xx = DeleteCases[xx, OR[u_, __] /; MemberQ[xx, u]];
    If[xx === {},
     ClearAttributes[OR, {Flat, Orderless}];
     SetAttributes[{NOT, OR}, HoldAll];
     Return["Failed, all members of the first argument are tautologies"]];
    variables =
     If[variables === All, LVariableSet[xx], Select[variables, MemberQ[xx, #, -1] &]];
    If[variables === {},
     ClearAttributes[OR, {Flat, Orderless}];
     SetAttributes[{NOT, OR}, HoldAll];
     Return["Failed, the first argument does not depend on given variables"]];
    variables1 = RemoveSubscript[variables];
    rules1 = Thread[Rule[variables, variables1]];
    variables = variables1;
    xx = xx /. rules1 // sort;
    {x1, x2, x3} = {{}, xx, xx};
    While[x2 =!= {} && FreeQ[x2, FALSE] ,
     AppendTo[sequence, x2];
     If[rules,
      variables1 = Select[variables, variableCase[x2, #] ≤ 1 &];
      While[variables1 =!= {},
       rules2 = Apply[Union, Map[variableRule[x2, #] &, variables1]];
       AppendTo[sequence, rules2];
       x2 = DeleteCases[x2 //. rules2, True] /. False → FALSE // sort;
       If[x2 =!= {} && FreeQ[x2, FALSE],
        AppendTo[sequence, x2];
        variables = Select[variables, MemberQ[x2, #, 3] &];
        variables1 = Select[variables, variableCase[x2, #] ≤ 1 &],
        variables1 = {}]]];
     If[x2 =!= {} && FreeQ[x2, FALSE],
      xx = generateResolvents[x1, x2, x3, reduce, sort, variables];
      xx = DeleteCases[xx /. False → FALSE,
           OR[u_, NOT[u_], ___]] //. OR[u_, u_] → u // sort;
      {x1, x2, x3} = {Join[x1, x3] // sort, xx, Join[x3, xx] // sort};
      variables = Select[variables, MemberQ[x2, #, -1] &]]];
    If[x2 === {} || MemberQ[x2, FALSE], AppendTo[sequence, x2]];
    sequence = sequence /. Reverse /@ rules1;
    sequence = sequence /. Thread[Rule[{False, True}, booleanValues]];
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    If[print === False,
     Column[sequence, Dividers → All, Spacings → 1], Print /@ Framed /@ sequence;]];
Protect[FullResolutionSequence];
```

ResolutionDepth[x, options] characterizes the complexity of the list x of clauses by the triple depth={±n,r,t}, where n is the length of the list FullResolutionSequence[x,options], r is the total number of clauses in it, and t is the CPU time spent in the *Mathematica* kernel. The sign is "+" if the list x is satisfiable, and "-" in the oppsite case.
Alternatives for options (default value = the first alternative) :
- Reduce -> True | False,

- Rules -> True | False,
- Sort -> DeleteDuplicates | Union | False,
- Trace -> True | False,
- Variables -> All | List of logical variables.

```
Unprotect[ResolutionDepth];
ClearAll[ResolutionDepth];
Attributes[ResolutionDepth] = {};
Options[ResolutionDepth] =
  {Reduce → True, Rules → True, Sort → False, Trace → True, Variables → All};

ResolutionDepth[x_, options___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
ResolutionDepth[x_, options___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
ResolutionDepth[x_, options___Rule] := Module[{case, depth, n = 0, reduce, rules,
    rules1, sort, timing, totalDepth, trace, variables, variables1, x1, x2, x3, xx},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
   {reduce, rules, sort, trace, variables} =
    {Reduce, Rules, Sort, Trace, Variables} /. {options} /. Options[ResolutionDepth];
   If[sort === False, sort = Identity];
   xx = DeleteCases[x, OR[u__, NOT[u__], ___]];
   xx = DeleteCases[xx, OR[u_, __] /; MemberQ[xx, u]]; If[xx === {},
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    Return["Failed, all members of the first argument are tautologies"]];
   variables =
    If[variables === All, LVariableSet[xx], Select[variables, ! FreeQ[xx, #] &]];
   If[variables === {},
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    Return["Failed, the first argument does not depend on given variables"]];
   variables1 = RemoveSubscript[variables];
   rules1 = Thread[Rule[variables, variables1]];
   variables = variables1;
   xx = xx /. rules1 // sort;
   {x1, x2, x3} = {{}, xx, xx};
   depth = {{0, Length[x2], 0}};
   While[x2 ≠ {} && FreeQ[x2, False],
    If[rules,
     While[timing =
       Timing[variables1 = Select[variables, variableCase[x2, #] ≤ 1 &];
        If[variables1 =!= {},
         rules1 = Apply[Union, Map[variableRule[x2, #] &, variables1]];
         x2 = DeleteCases[x2 //. rules1, True] // sort;
         variables = If[x2 =!= {} && FreeQ[x2, False],
           Select[variables, MemberQ[x2, #, -1] &], {}]]];
      variables1 =!= {},
```

```
      If[x2 =!= {}, depth = Append[depth,
         {If[FreeQ[x2, False], 1, -1] (n = n + 1), Length[x2], timing[[1]]}]]];
    If[x2 =!= {} && FreeQ[x2, False],
     timing = Timing[
       x2 = generateResolvents[x1, x2, x3, reduce, sort, variables];
       x2 = DeleteCases[x2, OR[u__, NOT[u__], ___]] /. OR[u_, u_] → u // sort;
       {x1, x2, x3} = {Join[x1, x3], x2, Join[x3, x2]};
       variables = Select[variables, MemberQ[x2, #, -1] &]];
     If[x2 =!= {}, depth = Append[depth,
        {If[FreeQ[x2, False], 1, -1] (n = n + 1), Length[x2], timing[[1]]}]]];
   If[x2 === {}, depth = Append[depth, {n = n + 1, 0, timing[[1]]}]];
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
   If[trace, depth, Prepend[Rest /@ depth // Total, depth[[-1, 1]]]]];
Protect[ResolutionDepth];
```

ResolutionSequence[x_List, options] tests whether the list x of clauses is satisfiable or not and in the positive case with the option Rules ->True finds, in some cases, values of logical variables for which all clauses in x are true. The algorithm used can be roughly described as a successive elimination of logical variables.
Alternatives for options (default value = the first alternative) :
- BooleanValues -> {False -> 0, True -> 1} | {False -> FalseSymbol, True -> TrueSymbol},
- ItemSize -> Automatic,
- Print -> False | True,
- Reduce -> True | False,
- Rules -> True | False,
- SelectionRule-> First | Last | Random,
- Sort -> DeleteDuplicates | Union | False,
- Spacings -> Automatic | Number,
- TableBreaks -> None | integer > 1 | increasing list of integers > 1,
- Variables -> All | List of logical variables.

```
Unprotect[ResolutionSequence];
ClearAll[ResolutionSequence];
Attributes[ResolutionSequence] = {};
Options[ResolutionSequence] = {BooleanValues → {0, 1}, ItemSize → Automatic,
    Print → False, Reduce → True, Rules → True, SelectionRule → First,
    Sort → DeleteDuplicates, Spacings → 1, SequenceBreaks → None, Variables → All};

ResolutionSequence[x_, options___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
ResolutionSequence[x_, options___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
ResolutionSequence[x_, options___Rule] := Module[
   {booleanValues, case, itemSize, print, reduce, row, rule, rules, rules1, rules2 = {},
    selectionRule, sequence = {}, sequenceBreaks, sequenceLength, sequenceParts,
    sort, spacings, v, variables, variables0, variables1, x0, x1, x2, x3, x12, xx},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
```

```
{booleanValues, itemSize, print, reduce,
  rules, selectionRule, sequenceBreaks, sort, spacings, variables} =
 {BooleanValues, ItemSize, Print, Reduce, Rules, SelectionRule,
     SequenceBreaks, Sort, Spacings, Variables} /. {options} /.
   Options[ResolutionSequence] /. Random → RandomChoice;
If[sort === False, sort = Identity];
xx = DeleteCases[x, OR[u__, NOT[u__], ___]];
xx = DeleteCases[xx, OR[u_, __] /; MemberQ[xx, u]];
If[xx === {},
 ClearAttributes[OR, {Flat, Orderless}];
 SetAttributes[{NOT, OR}, HoldAll];
 Return["Failed, all members of the first argument are tautologies"]];
variables =
 If[variables === All, LVariableSet[xx], Select[variables, MemberQ[xx, #, ∞] &]];
If[variables === {},
 ClearAttributes[OR, {Flat, Orderless}];
 SetAttributes[{NOT, OR}, HoldAll];
 Return["Failed, the first argument does not depend on given variables"]];
variables1 = RemoveSubscript[variables];
rules1 = Thread[Rule[variables, variables1]];
variables = {};
xx = xx /. rules1 // sort;
{variables0, x0} = {variables1, xx};
Label["x0"];
While[x0 =!= {} && FreeQ[x0, FALSE] && variables0 =!= {},
 v = selectionRule[variables0];
 {sequence, variables} = {Append[sequence, x0], Append[variables, v]};
 While[Length[sequence] ≥ 3 && sequence[[-1]] === sequence[[-3]],
  sequence = Drop[sequence, -2]];
 case = variableCase[x0, v];
 Which[case ≤ 1 && rules,
  rule = variableRule[x0, v];
  AppendTo[sequence, Row[{Global`•, rule // First, Global`•}]];
  x0 = DeleteCases[x0 //. rule, True] // sort;
  variables0 = Select[variables0, MemberQ[x0, #, ∞] &];
  rules2 = Join[rules2, rule],
  True,
  x1 = Select[x0, SameQ[# /. v → True, True] &];
  x2 = Select[x0, SameQ[# /. v → False, True] &];
  x3 = Select[x0, FreeQ[#, v] &];
  x12 = Distribute[OR[x1 /. v → False, x2 /. NOT[v] → False], List];
  x12 = DeleteCases[x12 /. False → FALSE, OR[u__, NOT[u__], ___]];
  x12 = x12 //. OR[u_, u_] → u // sort;
  If[reduce,
   x12 = DeleteCases[x12, OR[u_, __] | u_ /; MemberQ[x3, u]],
   x12 = DeleteCases[x12, u_ /; MemberQ[x3, u]]];
  AppendTo[sequence, Row[{" ", Global`•, v, Global`•}]];
  row = Which[x1 =!= {} && x2 =!= {} && x3 =!= {}, {{{x1, x2} → x12, x3}, ", "},
    x1 =!= {} && x2 =!= {} && x3 === {}, {{{x1, x2} → x12}},
    (x1 === {} || x2 === {}) && x3 =!= {}, {{{x1 ~ Join ~ x2}, x3}, ", "},
```

```
        (x1 === {} || x2 === {}) && x3 === {}, {{{x1 ~ Join ~ x2}}},
        x1 === {} && x2 === {} && x3 =!= {}, {{x3}}];
      If[(x1 =!= {} && x2 =!= {}) || x3 =!= {}, AppendTo[sequence, Row @@ row];
       x0 = Join[x12, x3] // sort];
      variables0 = Select[variables0, MemberQ[x0, #, ∞] && (# =!= v) &]]];
   AppendTo[sequence, x0];
   While[Length[sequence] ≥ 3 && sequence[[-1]] === sequence[[-3]],
    sequence = Drop[sequence, -2]];
   If[FreeQ[x0, FALSE] && rules2 =!= {}, AppendTo[sequence,
     Row[{Global`•, Global`•, rules2 // sort, Global`•, Global`•}]];
    x0 = xx //. rules2 // DeleteCases[#, True] & // sort;
    If[{} =!= x0 =!= xx, xx = x0;
     variables0 = Select[variables1, MemberQ[x0, #, ∞] &] // Reverse;
     Goto["x0"]]];
   sequence =
    sequence /. Reverse /@ rules1 //. Thread[Rule[{False, True}, booleanValues]];
   (* Formating output *)
   sequenceLength = Length@sequence;
   Which[sequenceBreaks ~ MatchQ ~ _Integer,
    sequenceBreaks = Range[sequenceBreaks + 1, sequenceLength - 1, sequenceBreaks],
    sequenceBreaks ~ MatchQ ~ {__Integer},
    sequenceBreaks = Select[sequenceBreaks + 1, # < sequenceLength &],
    True, sequenceBreaks = {}];
   sequenceBreaks = Union[{1}, sequenceBreaks, {sequenceLength + 1}];
   sequenceParts = {sequenceBreaks[[#]], sequenceBreaks[[# + 1]] - 1} & /@
     Range[Length[sequenceBreaks] - 1];
   sequenceParts = Take[sequence, #] & /@ sequenceParts;
   sequence =
    Column[#, Dividers → All, ItemSize → itemSize, Spacings → 1] & /@ sequenceParts;
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
   If[print === False,
    If[Length@sequence == 1, First@sequence, sequence], StylePrint /@ sequence;]];
Protect[ResolutionSequence];
```

ResolutionTable

ResolutionTable[x_List, options] tests whether the list x of clauses is satisfiable or not. It uses almost the same algorithm as ResolutionSequence but the result is presented in the form of a table.
Alternatives for options (default value = the first alternative) :
- BooleanValues -> {0, 1} | {False -> FalseSymbol, True -> TrueSymbol} | None,
- Dividers -> All | as for Grid,
- ItemSize -> Automatic | {{Automatic, {1.5}}, 1} | as for Grid,
- Print -> False | True,
- Reduce -> False | True,
- Rules -> False | True,
- SelectionRule -> First | Last | Random,
- Sort -> DeleteDuplicates | Union | False,
- TestResult -> False | True,
- TableBreaks -> None |integer > 1| increasing list of integers > 1,
- Transpose -> False | True,

- Variables -> All | List of logical variables.

```
Unprotect[ResolutionTable];
ClearAll[ResolutionTable];
Options[ResolutionTable] =
  {BooleanValues → {0, 1}, Dividers → All, ItemSize → Automatic, Print → False,
   Reduce → False, Rules → False, SelectionRule → First, Sort → DeleteDuplicates,
   TableBreaks → None, TestResult → False, Transpose → False, Variables → All};

ResolutionTable[x_, options___Rule] :=
  "Failed, the first argument is not a list of clauses" /; Head[x] =!= List;
ResolutionTable[x_, options___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
ResolutionTable[x_, options___Rule] := Module[{booleanValues, case, dividers, heading,
    itemSize, print, reduce, result, rule, rules, rules0 = {}, rules1, selectionRule,
    sort, table, table0 = {}, table1, tableBreaks, tableLength, tableParts, testResult,
    transpose, v, variables, variables0, variables1, x0, x1, x2, x3 = {}, xx},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
   {booleanValues, dividers, itemSize, print, reduce, rules,
     selectionRule, sort, tableBreaks, testResult, transpose, variables} =
    {BooleanValues, Dividers, ItemSize, Print, Reduce, Rules, SelectionRule,
        Sort, TableBreaks, TestResult, Transpose, Variables} /. {options} /.
      Options[ResolutionTable] /. Random → RandomChoice;
   If[sort === False, sort = Identity];
   xx = DeleteCases[x, OR[u__, NOT[u__], ___]];
   xx = DeleteCases[xx, OR[u_, __] /; MemberQ[xx, u]];
   If[xx === {},
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    Return["Failed, all members of the first argument are tautologies"]];
    variables =
    If[variables === All, LVariableSet[xx], Select[variables, ! FreeQ[xx, #] &]];
   If[variables === {},
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    Return["Failed, the first argument doesn't depend on given logical variables"]];

   variables1 = RemoveSubscript[variables];
   rules1 = Thread[Rule[variables, variables1]];
   variables = {};
   xx = xx /. rules1 // sort;
   {variables0, x0} = {variables1, xx};
   While[FreeQ[x0, FALSE] && variables0 =!= {},
    v = selectionRule[variables0];
    variables = Append[variables, v];
    case = variableCase[x0, v];
    Which[case ≤ 1 && rules, rule = variableRule[x0, v];
     AppendTo[xx, Row[{Global`• , rule // First, Global`•}]];
```

```
    x0 = DeleteCases[x0 //. rule, True] // sort;
    rules0 = Join[rules0, rule];
    xx = Join[xx, DeleteCases[x0, u_ /; MemberQ[xx, u]]],
    True,
    x1 = Select[x0, SameQ[# /. v → True, True] &];
    x2 = Select[x0, SameQ[# /. v → False, True] &];
    x3 = Select[x0, FreeQ[#, v] &];
    x0 = Distribute[OR[x1 /. v → False, x2 /. NOT[v] → False], List];
    x0 = DeleteCases[x0 /. False → FALSE, OR[u__, NOT[u__], ___]];
    x0 = x0 //. OR[u_, u_] → u // sort;
    If[reduce,
     x0 = DeleteCases[x0, OR[u_, __] /; MemberQ[x3, u]],
     x0 = DeleteCases[x0, u_ /; MemberQ[x3, u]]];
    AppendTo[xx, Row[{" ", Global`•, v, Global`•}]];
    xx = Join[xx, x0];
    x0 = Join[x0, x3] // sort];
  variables0 =
    If[testResult, variables1 = Select[variables1, (# =!= v) &];
     variables1, Select[variables0, (# =!= v) &]]];
 variables0 = Prepend[variables, ∞];
 heading = Prepend[variables /. Reverse /@ rules1, Global`■];
 table =
  Table[
   Function[u,
     If[Intersection[
         variableRule[u, #][[1, 2]] & /@ Take[variables0, n - 1], {False, True}] === {},
       variableRule[u, variables0[[n]]][[1, 2]], Global`•]] /@ xx,
    {n, 2, Length[variables0]}];
 table = Prepend[table, xx] // Transpose;

 If[FreeQ[x0, FALSE] && testResult,
  table = Append[table, variables0 /. ∞ → Global`■ /. rules0];
  {variables0, x0} =
    {Prepend[variables, Global`□] /. rules0, Select[xx, ClauseQ[#] &]};
  If[rules0 =!= {}, x0 = DeleteCases[x0 /. rules0, True]];
  variables = Select[variables, MemberQ[x0, #, ∞] && FreeQ[{∞, True, False}, #] &];
  While[x0 =!= {} && variables =!= {},
    variables1 = Select[variables, variableCase[x0, #] ≤ 1 &];
    If[variables1 =!= {},
      v = selectionRule[variables1]; rule = First@variableRule[x0, v],
      v = Last[variables]; rule = v → True];
    AppendTo[rules0, rule];
    x0 = DeleteCases[x0 /. rule, True]; variables0 = variables0 /. rule;
    variables = DeleteCases[variables, v];
    table = Append[table, variables0]];
  result = Select[xx, ClauseQ[#] &] /. rules0 // AND @@ # &,
  testResult = False];

 table = table /. Reverse /@ rules1 //. Thread[Rule[{False, True}, booleanValues]];
 tableLength = Length[table];
```

```
    Which[tableBreaks ~MatchQ~ _Integer,
     tableBreaks = Range[tableBreaks + 1, tableLength - 1, tableBreaks],
     tableBreaks ~MatchQ~ {__Integer},
     tableBreaks = Select[tableBreaks + 1, # < tableLength &],
     True, tableBreaks = {}];
    tableBreaks = Union[{1}, tableBreaks, {tableLength + 1}];
    tableParts =
     {tableBreaks[[#]], tableBreaks[[# + 1]] - 1} & /@ Range[Length[tableBreaks] - 1];
    tableParts = Take[table, #] & /@ tableParts;
    tableParts = Prepend[#, heading] & /@ tableParts;
    If[transpose, tableParts = Transpose /@ tableParts];
    table = Grid[#, Dividers → dividers, ItemSize → itemSize] & /@ tableParts;
    If[testResult,
     table = ReplacePart[table, -1 → Column[{table[[-1]], result}, Center]]];
    ClearAttributes[OR, {Flat, Orderless}];
    SetAttributes[{NOT, OR}, HoldAll];
    If[ print === False,
     If[Length@table == 1, First@table, table], StylePrint /@ table;]];
Protect[ResolutionTable];
```

Preliminaries for RefutationTree

BTreeQ[x] tests whether x is a binary tree.

```
Unprotect[BT, BTreeQ];
Clear[BT, BTreeQ];
BTreeQ[{BT[x_]}] := True;
BTreeQ[{BT[x_], y_, z_}] := And[BTreeQ[y], BTreeQ[z]];
BTreeQ[x_] := False;
Notation[ [[x_]] ⟺ BT[x_] ];
Protect[BT, BTreeQ];
```

BTreeForm prints a binary tree as a table or a tree.
Alternatives for options (default value = the first alternative):
- Format -> TreePlot | TableForm | List,
- RootPosition -> Right | Left | Top | Bottom,
- TableSpacing -> {nonnegative integer, nonnegative integer},
- Format ->List | TreePlot | TableForm.

```
Unprotect[BTreeForm];
ClearAll[BTreeForm];
Options[BTreeForm] =
  {AspectRatio → Automatic, DirectedEdges → True, Format → TableForm,
   Frame → True, ImagePadding → All, ImageSize → {500, Automatic},
   PlotTheme → "ClassicLabeled", RootPosition → Right, TableSpacing → {0, 0}};

BTreeForm[x_ /; ! BTreeQ[x], opts___Rule] := "Failed, the argument is not binary tree."
BTreeForm[x_ /; BTreeQ[x], opts___Rule] :=
  Module[{bTree, bTreeDepth, format, frame, function, nodes, options, plotTheme,
```

```
     position, root, rootPosition, rules1, rules2, rules3, spacing, xx},
    {format, frame, rootPosition, root, spacing} =
     {Format, Frame, RootPosition, Root, TableSpacing} /. {opts} /. Options[BTreeForm];
    options = Union[{opts}, FilterRules[Options[BTreeForm], Except[{opts}]]];
    options = FilterRules[options, Options[TreePlot]];
    root = x[[1, 1]];
    Which[format === TableForm,
     {bTree = bTreeForm[x, rootPosition], bTreeDepth = Depth[bTree]};
     spacing = Table[spacing, {bTreeDepth}] // Flatten;
     bTree = TableForm[bTree,
       TableAlignments → Table[rootPosition, {bTreeDepth}], TableSpacing → spacing];
     If[frame, Framed[bTree, FrameMargins → 10], bTree],
     format === List || format === TreePlot,
     {xx = x //. BT → List, nodes = xx // Flatten};
     position = Position[xx, {#}, ∞];
     function = Function[u, Rule[#, u]];
     rules1 = ({position = Position[xx, {#}, ∞], function = Function[u, Rule[#, u]]};
             If[Length[position] == 1, Rule[#, position], function /@ position]) & /@
           nodes //. {{u__}} → {u} // Flatten // Union;
     rules2 = Join[Select[Gather[rules1, #1[[1]] == #2[[1]] &], Length[#] == 1 &],
        Map[MapIndexed[Rule[{#1[[1]]}#2[[1]], #1[[2]]] &, #] &,
         Select[Gather[rules1, #1[[1]] == #2[[1]] &], Length[#] > 1 &]]] // Flatten;
     rules3 = Reverse /@ rules2 /. Rule[u_, v_] → Rule[u, {v}];
     xx = ReplacePart[xx, rules3];
     xx = xx //. {{u_}, {{v1_}, w1___}, {{v2_}, w2___}} :→
        {Rule[u, v1], {{v1}, w1}, Rule[u, v2], {{v2}, w2}};
     xx = Cases[xx, _Rule, ∞] /. Rule[u_, v_] → Rule[v, u];
     If[format === List, xx,
      TreePlot[xx, rootPosition, root, options]]]];
Protect[BTreeForm];

Unprotect[bTreeForm];
Clear[bTreeForm];
bTreeForm[{BT[x_]}, Left | Right | Top | Bottom] := {{x}};
bTreeForm[{BT[x_], y_, z_}, Left] := {{" ", bTreeForm[y, Left]},
   {SequenceForm["•", x, "• "], " "}, {" ", bTreeForm[z, Left]}};
bTreeForm[{BT[x_], y_, z_}, Right] :=
  {{bTreeForm[y, Right], " "}, {" ", SequenceForm[" •", x, "•"]},
   {bTreeForm[z, Right], " "}};
bTreeForm[{BT[x_], y_, z_}, Top] :=
  {{" ", SequenceForm["•", x, "•"], " "}, {bTreeForm[y, Top], " ",
    bTreeForm[z, Top]}};
bTreeForm[{BT[x_], y_, z_}, Bottom] :=
  {{bTreeForm[y, Bottom], " ", bTreeForm[z, Bottom]},
   {" ", SequenceForm["•", x, "•"], " "}};
Protect[bTreeForm];
```

Refutation Tree

RTree[x] decides whether the list x of clauses not containing commands is satisfiable or not, and in the latter case it finds a refutation tree by an algorithm described in Anthony Galton : Logic for

Information Technology, John Wiley & Sons, 1990, pp.104 - 105.
Alternatives for Options (default value = the first alternative) :
SelectionRule -> First | Last | Random

```
Unprotect[RTree];
ClearAll[RTree];
Options[RTree] = {SelectionRule → First};

RTree[x_, options___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
RTree[x_, options___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
RTree[{}, options___Rule] = $Failed;
RTree[{FALSE}, ___Rule] = {BT[FALSE]};
RTree[{x_?ClauseQ}, options___Rule] := {BT[x]};
RTree[{x_, NOT[x_]}, options___Rule] = {BT[FALSE], {BT[x]}, {BT[NOT[x]]}};
RTree[x_, options___Rule] :=
  Module[{selectionRule, tree},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
   selectionRule =
    SelectionRule /. {options} /. Options[RTree] /. Random → RandomChoice;
   tree = rTree[x, SelectionRule → selectionRule];
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
   tree];
Protect[RTree];
```

```
Unprotect[rTree];
ClearAll[rTree];
Options[rTree] = {SelectionRule → First};

rTree[{}, options___Rule] = $Failed;
rTree[{x_, NOT[x_]}, options___Rule] = {BT[FALSE], {BT[x]}, {BT[NOT[x]]}};
rTree[x_, options___Rule] :=
  Module[
   {clause, rule1, rule2, selectionRule, tree, tree1, u, v, w, w1, y, y1, variables, xx},
   selectionRule = SelectionRule /. {options} /. Options[rTree] /.
     Random → RandomChoice;
   rule1 := ({BT[u_]} /; MemberQ[x, OR[u, NOT[v]], Infinity]) → {BT[OR[u, NOT[v]]]};
   rule2 := ({BT[u_], {BT[w_], w1___}, {BT[y_], y1___}} /;
        And[Not[FreeQ[{w, y}, NOT[v]]], FreeQ[{u}, NOT[v]]]) →
      {BT[OR[u, NOT[v]]], {BT[w], w1}, {BT[y], y1}} //. OR[u_, u_] → u;
   variables = Select[LVariableSet[x], And[MemberQ[x, #], MemberQ[x, NOT[#]]] &];
   Catch[
    If[variables =!= {}, v = selectionRule[variables];
     Throw[{BT[FALSE], {BT[v]}, {BT[NOT[v]]}}]];
    xx = Select[x, FreeQ[#, NOT] &];
    If[Or[xx === {}, xx === x], Throw[$Failed]];
```

```
    {clause, xx, tree, variables} =
     {#, DeleteCases[x, #], {BT[#]}, LVariableSet[#]} &[selectionRule[xx]];
    While[variables =!= {},
     {clause, v} = {If[variables === {#}, FALSE, DeleteCases[clause, #]], #} &[
       selectionRule[variables]];
     If[MemberQ[xx, NOT[v]],
      tree = {BT[clause], tree, {BT[NOT[v]]}},
      tree1 = rTree[Map[DeleteCases[#, NOT[v]] &, xx], options];
      If[tree1 =!= $Failed,
       tree1 = tree1 //. rule1 //. rule2 //. OR[FALSE, u__] → OR[u];
       If[Part[tree1, 1] === BT[FALSE], Throw[rTree[xx, options]],
        tree = {BT[clause], tree, tree1}],
       Throw[$Failed]]];
     variables = DeleteCases[variables, v]];
    Throw[tree]]];
Protect[rTree];
```

RefutationTree[x] outputs RTree[x] in a convenient graphic form.
Alternatives for Options (default value = the first alternative):
- Format -> TreePlot | TableForm | List,
- RootPosition -> Right | Left | Top | Bottom,
- SelectionRule -> First | Last | RandomChoice,
- Sort -> DeleteDuplicates | Union,
- TableSpacing -> Automatic | {nonnegative integer, nonnegative integer}.

```
Unprotect[RefutationTree];
ClearAll[RefutationTree];
Options[RefutationTree] = {AspectRatio → Automatic, DirectedEdges → True,
   Format → TreePlot, Frame → True, ImageSize → {500, Automatic}, ImagePadding → All,
   PlotTheme → "ClassicLabeled", RootPosition → Right, Root → FALSE,
   SelectionRule → First, Sort → DeleteDuplicates, TableSpacing → {0, 0}};

RefutationTree[x_, opts___Rule] :=
  "Failed, the first argument is not a list" /; Head[x] =!= List;
RefutationTree[x_, opts___Rule] :=
  "Failed, a member of the first argument is not clause" /;
   MemberQ[Map[ClauseQ, x], False];
RefutationTree[x_, opts___Rule] :=
  Module[{options, selectionRule, sort, xx},
   ClearAttributes[{NOT, OR}, HoldAll];
   SetAttributes[OR, {Flat, Orderless}];
   {selectionRule, sort} =
    {SelectionRule, Sort} /. {opts} /. Options[RefutationTree];
   options = Union[{opts}, FilterRules[Options[RefutationTree], Except[{opts}]]];
   options = FilterRules[options, Options[BTreeForm]];
   xx = DeleteCases[x, OR[y_, NOT[y_], z___]];
   xx = xx //. OR[y_, y_] → y // sort;
   xx = rTree[xx, SelectionRule → selectionRule];
   ClearAttributes[OR, {Flat, Orderless}];
   SetAttributes[{NOT, OR}, HoldAll];
```

```
    If[xx === $Failed, Return["The set of clauses is satisfiable"]];
    BTreeForm[xx, Sequence @@ options]];
Protect[RefutationTree];
```

End Private Context

```
End[];
```

EndPackage

```
EndPackage[];
```

Connectives Palette

```
NotebookPut[
  Notebook[{Cell[
     BoxData[
      GridBox[{{ButtonBox["¬"], ButtonBox["∧"], ButtonBox["∨"], ButtonBox["⇒"],
         ButtonBox["↑"], ButtonBox["↓"], ButtonBox["⊕"], ButtonBox["⇔"]}}]],
     "NotebookDefault", PageBreakAbove → True, FontFamily → "Courier",
     FontWeight → "Plain"]},
   NotebookAutoSave → True, ClosingSaveDialog → False, Editable → True,
   Selectable → False, Clickable → False, WindowToolbars → {}, Deletable → True,
   WindowSize → {Fit, Fit}, WindowMargins → {{Automatic, 140}, {Automatic, 2}},
   WindowFrame → "Palette", WindowElements → {}, WindowFrameElements → "CloseBox",
   WindowClickSelect → False, WindowTitle → "Connectives", ShowCellBracket → False,
   CellMargins → {{10, 10}, {5, 5}}, Enabled → True, CellOpen → True,
   ShowCellLabel → False, ShowCellTags → False, Magnification → 1.,
   FrontEndVersion → "8.0 for Microsoft Windows (32-bit) (October 6, 2011)",
   StyleDefinitions → "Palette.nb"]];
```

Names

```
Names["Notation`*"]
```

```
{Action, ActiveInputAliases, AddInputAlias, AutoLoadNotationPalette,
 ClearNotations, CreateNotationRules, InfixNotation, Notation, NotationBoxTag,
 NotationMadeBoxesTag, NotationMakeBoxes, NotationMakeExpression, NotationPatternTag,
 ParsedBoxWrapper, PrintNotationRules, RemoveInfixNotation, RemoveNotation,
 RemoveNotationRules, RemoveSymbolize, Symbolize, SymbolizeRootName, WorkingForm}
```

```
Names["PropositionalLogic`*"]
```

```
{AllFalse, AND, BooleanValues, BT, BTreeForm, BTreeQ, ClauseQ, CNF, Connectives,
 ConvertFormula, DNF, EmptyTruthTable, EQUIV, FALSE, FullResolutionSequence, HList,
 IMPLIES, Labels, LastFalse, LastTrue, LAtomQ, LEquivalentQ, LFormulaQ, LiteralQ,
 LStringQ, LTautologyQ, LVariableSet, Mixed, NAND, NOR, NormalizeLString, NOT,
 OnlyLastFalse, OR, PropositionalLogic, ReduceSpaces, RefutationTree, ReleaseHE,
 ResolutionDepth, ResolutionSequence, ResolutionTable, Resolvents, ReverseValues,
 RootPosition, RTree, Rules, SelectionRule, SelectValuations, SequenceBreaks, TableBreaks,
 TestResult, ToClauses, ToLFormula, ToLString, TRUE, TruthTable, x, XOR, y, y$}
```

```
StringDrop[#, 27] & /@ Names["PropositionalLogic`Private`*"]
```

```
{alignment, alignment$, booleanValues, booleanValues$, bTree, bTreeDepth, bTreeDepth$,
 bTreeForm, bTree$, case, case$, clause, clause$, Complete, depth, depth$,
 dividers, dividers$, f, form, format, format$, frame, frame$, function, function$,
 generateResolvents, heading, heading$, InfixConnectives, InfixStrings, itemSize,
 itemSize$, labels, labels$, Minimal, n, nodes, nodes$, NormalizeLString, n$, options,
 options$, opts, p, plotTheme, plotTheme$, position, position0, position0$, position1,
 position1$, positions, positions$, position$, PrefixConnectives, PrefixStrings, print,
 print$, p$, reduce, reduce$, RemoveSubscript, RestoreSubscript, result, result$,
 reverseValues, reverseValues$, root, rootPosition, rootPosition$, root$, row, row$,
 rTree, rule, rule1, rule1$, rule2, rule2$, rules, rules0, rules0$, rules1, rules1$,
 rules2, rules2$, rules3, rules3$, rules$, rule$, s, selectionRule, selectionRule$,
 selectValuations, selectValuations$, sequence, sequenceBreaks, sequenceBreaks$,
 sequenceLength, sequenceLength$, sequenceParts, sequenceParts$, sequence$, sort, sort$,
 spacing, spacings, spacings$, spacing$, ss1, ss1$, ss2, ss2$, ss3, ss3$, ssVariables1,
 ssVariables1$, ssVariables2, ssVariables2$, string, Strings, SubscriptBoxToString,
 t, table, table0, table0$, table1, table1$, tableBreaks, tableBreaks$, tableLength,
 tableLength$, tableParts, tableParts$, tableWidth, tableWidth$, table$, testResult,
 testResult$, timing, timing$, ToLettersAndDigits, totalDepth, totalDepth$, trace,
 trace$, transpose, transpose$, tree, tree1, tree1$, tree$, tuples, tuples$, type,
 type$, u, u$, v, v1, v1$, v2, v2$, variableCase, variableRule, variables, variables0,
 variables0$, variables1, variables1$, variablesLength, variablesLength$,
 variables$, varsnumber, varsnumber$, v$, w, w1, w1$, w2, w2$, w$, x0, x0$, x1, x10,
 x10$, x11, x11$, x12, x12$, x1$, x2, x20, x20$, x21, x21$, x22, x22$, x2$, x3, x3$,
 xx, xx1, xx12, xx12$, xx1$, xx21, xx21$, xx22, xx22$, xxx, xxx$, xx$, y1, y1$, z}
```