Sage Reference Manual: Symbolic Logic

Release 7.3

The Sage Development Team

Aug 05, 2016

CONTENTS

CHAPTER

ONE

PROPOSITIONAL CALCULUS

Formulas consist of the following operators:

- `&` – and
- `|` – or
- `~` – not
- `^` – xor
- `->` – if-then
- `<->` – if and only if

Operators can be applied to variables that consist of a leading letter and trailing underscores and alphanumerics. Parentheses may be used to explicitly show order of operation.

AUTHORS:

- Chris Gorecki (2006): initial version, propcalc, boolformula, logictable, logicparser, booleval
- Michael Greenberg – boolopt
- Paul Scurek (2013-08-05): updated docstring formatting
- Paul Scurek (2013-08-12): added *get_formulas()*, *consistent()*, *valid_consequence()*

EXAMPLES:

We can create boolean formulas in different ways:

```
sage: f = propcalc.formula("a&((b|c)^a->c)<->b")
sage: g = propcalc.formula("boolean<->algebra")
sage: (f&~g).ifthen(f)
((a&((b|c)^a->c)<->b)&(~(boolean<->algebra)))->(a&((b|c)^a->c)<->b)
```

We can create a truth table from a formula:

```
sage: f.truthtable()
a      b      c      value
False  False  False  True
False  False  True   True
False  True   False  False
False  True   True   False
True   False  False  True
True   False  True   False
True   True   False  True
True   True   True   True
sage: f.truthtable(end=3)
```

```
a      b      c      value
False  False  False  True
False  False  True   True
False  True   False  False
sage: f.truthtable(start=4)
a      b      c      value
True   False  False  True
True   False  True   False
True   True   False  True
True   True   True   True
sage: propcalc.formula("a").truthtable()
a      value
False  False
True   True
```

Now we can evaluate the formula for a given set of input:

```
sage: f.evaluate({'a':True, 'b':False, 'c':True})
False
sage: f.evaluate({'a':False, 'b':False, 'c':True})
True
```

And we can convert a boolean formula to conjunctive normal form:

```
sage: f.convert_cnf_table()
sage: f
(a|~b|c)&(a|~b|~c)&(~a|b|~c)
sage: f.convert_cnf_recur()
sage: f
(a|~b|c)&(a|~b|~c)&(~a|b|~c)
```

Or determine if an expression is satisfiable, a contradiction, or a tautology:

```
sage: f = propcalc.formula("a|b")
sage: f.is_satisfiable()
True
sage: f = f & ~f
sage: f.is_satisfiable()
False
sage: f.is_contradiction()
True
sage: f = f | ~f
sage: f.is_tautology()
True
```

The equality operator compares semantic equivalence:

```
sage: f = propcalc.formula("(a|b)&c")
sage: g = propcalc.formula("c&(b|a)")
sage: f == g
True
sage: g = propcalc.formula("a|b&c")
sage: f == g
False
```

It is an error to create a formula with bad syntax:

```
sage: propcalc.formula("")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&b~(c|(d)")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&&b")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&b a")
Traceback (most recent call last):
...
SyntaxError: malformed statement

It is also an error to not abide by the naming conventions.
sage: propcalc.formula("~a&9b")
Traceback (most recent call last):
...
NameError: invalid variable name 9b: identifiers must begin with a letter and contain␣
 ↪only alphanumerics and underscores
```

sage.logic.propcalc. **consistent** (**formulas*)

Determine if the formulas are logically consistent.

INPUT:

- `*formulas` – instances of `BooleanFormula`

OUTPUT:

A boolean value to be determined as follows:

- `True` - if the formulas are logically consistent
- `False` - if the formulas are not logically consistent

EXAMPLES:

This example illustrates determining if formulas are logically consistent.

```
sage: f, g, h, i = propcalc.get_formulas("a<->b", "~b->~c", "d|g", "c&a")
sage: propcalc.consistent(f, g, h, i)
True
```

```
sage: j, k, l, m = propcalc.get_formulas("a<->b", "~b->~c", "d|g", "c&~a")
sage: propcalc.consistent(j, k ,l, m)
False
```

AUTHORS:

- Paul Scurek (2013-08-12)

sage.logic.propcalc. **formula** (*s*)

Return an instance of `BooleanFormula` .

INPUT:

- `s` – a string that contains a logical expression

OUTPUT:

An instance of `BooleanFormula`.

EXAMPLES:

This example illustrates ways to create a boolean formula:

```
sage: f = propcalc.formula("a&~b|c")
sage: g = propcalc.formula("a^c<->b")
sage: f&g|f
((a&~b|c)&(a^c<->b))|(a&~b|c)
```

We now demonstrate some possible errors:

```
sage: propcalc.formula("((a&b)")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("_a&b")
Traceback (most recent call last):
...
NameError: invalid variable name _a: identifiers must begin with a letter and␣
↪contain only alphanumerics and underscores
```

sage.logic.propcalc. **get_formulas** (**statements*)

Convert statements and parse trees into instances of `BooleanFormula`.

INPUT:

- `*statements` – strings or lists; a list must be a full syntax parse tree of a formula, and a string must be a string representation of a formula

OUTPUT:

The converted formulas in a list.

EXAMPLES:

This example illustrates converting strings into boolean formulas.

```
sage: f = "a&(~c<->d)"
sage: g = "d|~~b"
sage: h = "~(a->c)<->(d|~c)"
sage: propcalc.get_formulas(f, g, h)
[a&(~c<->d), d|~~b, ~(a->c)<->(d|~c)]
```

```
sage: A, B, C = propcalc.get_formulas("(a&b)->~c", "c", "~(a&b)")
sage: A
(a&b)->~c
sage: B
c
sage: C
~(a&b)
```

We can also convert parse trees into formulas.

```
sage: t = ['a']
sage: u = ['~', ['|', ['&', 'a', 'b'], ['~', 'c']]]
sage: v = "b->(~c<->d)"
sage: formulas= propcalc.get_formulas(t, u, v)
```

```
sage: formulas[0]
a
sage: formulas[1]
~((a&b)|~c)
sage: formulas[2]
b->(~c<->d)
```

AUTHORS:

- Paul Scurek (2013-08-12)

sage.logic.propcalc.**valid_consequence**(*consequence*, **formulas*)

Determine if `consequence` is a valid consequence of the set of formulas in `*formulas`.

INPUT:

- `*formulas` – instances of `BooleanFormula`
- `consequence` – instance of `BooleanFormula`

OUTPUT:

A boolean value to be determined as follows:

- `True` - if `consequence` is a valid consequence of the set of `*formulas`
- `False` - if `consequence is not a valid consequence of the set of ``*formulas`

EXAMPLES:

This example illustrates determining if a formula is a valid consequence of a set of other formulas:

```
sage: f, g, h, i = propcalc.get_formulas("a&~b", "c->b", "c|e", "e&a")
sage: propcalc.valid_consequence(i, f, g, h)
True
```

```
sage: j = propcalc.formula("a&~e")
sage: propcalc.valid_consequence(j, f, g, h)
False
```

```
sage: k = propcalc.formula("((p<->q)&r)->~c")
sage: propcalc.valid_consequence(k, f, g, h)
True
```

AUTHORS:

- Paul Scurek (2013-08-12)

CHAPTER

TWO

BOOLEAN FORMULAS

Formulas consist of the operators `&` , `|` , `~` , `^` , `->` , `<->` , corresponding to `and` , `or` , `not` , `xor` , `if...then` , `if and only if` . Operators can be applied to variables that consist of a leading letter and trailing underscores and alphanumerics. Parentheses may be used to explicitly show order of operation.

EXAMPLES:

Create boolean formulas and combine them with *`ifthen()`* method:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a&((b|c)^a->c)<->b")
sage: g = propcalc.formula("boolean<->algebra")
sage: (f&~g).ifthen(f)
((a&((b|c)^a->c)<->b)&(~(boolean<->algebra)))->(a&((b|c)^a->c)<->b)
```

We can create a truth table from a formula:

```
sage: f.truthtable()
a      b      c      value
False  False  False  True
False  False  True   True
False  True   False  False
False  True   True   False
True   False  False  True
True   False  True   False
True   True   False  True
True   True   True   True
sage: f.truthtable(end=3)
a      b      c      value
False  False  False  True
False  False  True   True
False  True   False  False
sage: f.truthtable(start=4)
a      b      c      value
True   False  False  True
True   False  True   False
True   True   False  True
True   True   True   True
sage: propcalc.formula("a").truthtable()
a      value
False  False
True   True
```

Now we can evaluate the formula for a given set of inputs:

```
sage: f.evaluate({'a':True, 'b':False, 'c':True})
False
sage: f.evaluate({'a':False, 'b':False, 'c':True})
True
```

And we can convert a boolean formula to conjunctive normal form:

```
sage: f.convert_cnf_table()
sage: f
(a|~b|c)&(a|~b|~c)&(~a|b|~c)
sage: f.convert_cnf_recur()
sage: f
(a|~b|c)&(a|~b|~c)&(~a|b|~c)
```

Or determine if an expression is satisfiable, a contradiction, or a tautology:

```
sage: f = propcalc.formula("a|b")
sage: f.is_satisfiable()
True
sage: f = f & ~f
sage: f.is_satisfiable()
False
sage: f.is_contradiction()
True
sage: f = f | ~f
sage: f.is_tautology()
True
```

The equality operator compares semantic equivalence:

```
sage: f = propcalc.formula("(a|b)&c")
sage: g = propcalc.formula("c&(b|a)")
sage: f == g
True
sage: g = propcalc.formula("a|b&c")
sage: f == g
False
```

It is an error to create a formula with bad syntax:

```
sage: propcalc.formula("")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&b~(c|(d)")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&&b")
Traceback (most recent call last):
...
SyntaxError: malformed statement
sage: propcalc.formula("a&b a")
Traceback (most recent call last):
...
SyntaxError: malformed statement
```

It is also an error to not abide by the naming conventions:

```
sage: propcalc.formula("~a&9b")
Traceback (most recent call last):
...
NameError: invalid variable name 9b: identifiers must begin with a letter and contain␣
↪only alphanumerics and underscores
```

AUTHORS:

- Chris Gorecki (2006): initial version
- Paul Scurek (2013-08-03): added polish_notation, full_tree, updated docstring formatting
- Paul Scurek (2013-08-08): added *implies()*

class sage.logic.boolformula. **BooleanFormula** (*exp*, *tree*, *vo*)
Bases: object

Boolean formulas.

INPUT:

- self – calling object
- exp – a string; this contains the boolean expression to be manipulated
- tree – a list; this contains the parse tree of the expression.
- vo – a list; this contains the variables in the expression, in the order that they appear; each variable only occurs once in the list

add_statement (*other*, *op*)
Combine two formulas with the given operator.

INPUT:

- other – instance of *BooleanFormula* ; this is the formula on the right of the operator
- op – a string; this is the operator used to combine the two formulas

OUTPUT:

The result as an instance of *BooleanFormula* .

EXAMPLES:

This example shows how to create a new formula from two others:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b")
sage: f = propcalc.formula("c^d")
sage: s.add_statement(f, '|')
(a&b)|(c^d)

sage: s.add_statement(f, '->')
(a&b)->(c^d)
```

convert_cnf ( )
Convert boolean formula to conjunctive normal form.

OUTPUT:

An instance of *BooleanFormula* in conjunctive normal form.

EXAMPLES:

This example illustrates how to convert a formula to cnf:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a ^ b <-> c")
sage: s.convert_cnf()
sage: s
(a|b|~c)&(a|~b|c)&(~a|b|c)&(~a|~b|~c)
```

We now show that *convert_cnf()* and *convert_cnf_table()* are aliases:

```
sage: t = propcalc.formula("a ^ b <-> c")
sage: t.convert_cnf_table(); t
(a|b|~c)&(a|~b|c)&(~a|b|c)&(~a|~b|~c)
sage: t == s
True
```

Note: This method creates the cnf parse tree by examining the logic table of the formula. Creating the table requires $O(2^n)$ time where n is the number of variables in the formula.

---

convert_cnf_recur ( )

Convert boolean formula to conjunctive normal form.

OUTPUT:

An instance of *BooleanFormula* in conjunctive normal form.

EXAMPLES:

This example hows how to convert a formula to conjunctive normal form:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a^b<->c")
sage: s.convert_cnf_recur()
sage: s
(~a|a|c)&(~b|a|c)&(~a|b|c)&(~b|b|c)&(~c|a|b)&(~c|~a|~b)
```

Note: This function works by applying a set of rules that are guaranteed to convert the formula. Worst case the converted expression has an $O(2^n)$ increase in size (and time as well), but if the formula is already in CNF (or close to) it is only $O(n)$.

This function can require an exponential blow up in space from the original expression. This in turn can require large amounts of time. Unless a formula is already in (or close to) being in cnf *convert_cnf()* is typically preferred, but results can vary.

---

convert_cnf_table ( )

Convert boolean formula to conjunctive normal form.

OUTPUT:

An instance of *BooleanFormula* in conjunctive normal form.

EXAMPLES:

This example illustrates how to convert a formula to cnf:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a ^ b <-> c")
sage: s.convert_cnf()
```

```
sage: s
(a|b|~c)&(a|~b|c)&(~a|b|c)&(~a|~b|~c)
```

We now show that *convert_cnf()* and *convert_cnf_table()* are aliases:

```
sage: t = propcalc.formula("a ^ b <-> c")
sage: t.convert_cnf_table(); t
(a|b|~c)&(a|~b|c)&(~a|b|c)&(~a|~b|~c)
sage: t == s
True
```

Note: This method creates the cnf parse tree by examining the logic table of the formula. Creating the table requires $O(2^n)$ time where n is the number of variables in the formula.

convert_expression ( )

Convert the string representation of a formula to conjunctive normal form.

EXAMPLES:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a^b<->c")
sage: s.convert_expression(); s
a^b<->c
```

convert_opt (*tree*)

Convert a parse tree to the tuple form used by `bool_opt()` .

INPUT:

- `tree` – a list; this is a branch of a parse tree and can only contain the '&', 'I' and '~' operators along with variables

OUTPUT:

A 3-tuple.

EXAMPLES:

This example illustrates the conversion of a formula into its corresponding tuple:

```
sage: import sage.logic.propcalc as propcalc, sage.logic.logicparser as␣
↪logicparser
sage: s = propcalc.formula("a&(b|~c)")
sage: tree = ['&', 'a', ['|', 'b', ['~', 'c', None]]]
sage: logicparser.apply_func(tree, s.convert_opt)
('and', ('prop', 'a'), ('or', ('prop', 'b'), ('not', ('prop', 'c'))))
```

Note: This function only works on one branch of the parse tree. To apply the function to every branch of a parse tree, pass the function as an argument in *apply_func()* in *logicparser* .

dist_not (*tree*)

Distribute '~' operators over '&' and 'I' operators.

INPUT:

- `tree` a list; this represents a branch of a parse tree

OUTPUT:

A new list.

EXAMPLES:

This example illustrates the distribution of '~' over '&':

```
sage: import sage.logic.propcalc as propcalc, sage.logic.logicparser as␣
↪logicparser
sage: s = propcalc.formula("~(a&b)")
sage: tree = ['~', ['&', 'a', 'b'], None]
sage: logicparser.apply_func(tree, s.dist_not) #long time
['|', ['~', 'a', None], ['~', 'b', None]]
```

Note: This function only operates on a single branch of a parse tree. To apply the function to an entire parse tree, pass the function as an argument to *apply_func()* in *logicparser*.

dist_ors (*tree*)

Distribute 'l' over '&'.

INPUT:

- `tree` – a list; this represents a branch of a parse tree

OUTPUT:

A new list.

EXAMPLES:

This example illustrates the distribution of 'l' over '&':

```
sage: import sage.logic.propcalc as propcalc, sage.logic.logicparser as␣
↪logicparser
sage: s = propcalc.formula("(a&b)|(a&c)")
sage: tree = ['|', ['&', 'a', 'b'], ['&', 'a', 'c']]
sage: logicparser.apply_func(tree, s.dist_ors) #long time
['&', ['&', ['|', 'a', 'a'], ['|', 'b', 'a']], ['&', ['|', 'a', 'c'], ['|', 'b
↪', 'c']]]
```

Note: This function only operates on a single branch of a parse tree. To apply the function to an entire parse tree, pass the function as an argument to *apply_func()* in *logicparser*.

equivalent (*other*)

Determine if two formulas are semantically equivalent.

INPUT:

- `self` – calling object
- `other` – instance of BooleanFormula class.

OUTPUT:

A boolean value to be determined as follows:

True - if the two formulas are logically equivalent

False - if the two formulas are not logically equivalent

EXAMPLES:

This example shows how to check for logical equivalence:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("(a|b)&c")
sage: g = propcalc.formula("c&(a|b)")
sage: f.equivalent(g)
True

sage: g = propcalc.formula("a|b&c")
sage: f.equivalent(g)
False
```

evaluate (*var_values*)

Evaluate a formula for the given input values.

INPUT:

- `var_values` – a dictionary; this contains the pairs of variables and their boolean values.

OUTPUT:

The result of the evaluation as a boolean.

EXAMPLES:

This example illustrates the evaluation of a boolean formula:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a&b|c")
sage: f.evaluate({'a':False, 'b':False, 'c':True})
True
sage: f.evaluate({'a':True, 'b':False, 'c':False})
False
```

full_tree ( )

Return a full syntax parse tree of the calling formula.

OUTPUT:

The full syntax parse tree as a nested list

EXAMPLES:

This example shows how to find the full syntax parse tree of a formula:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a->(b&c)")
sage: s.full_tree()
['->', 'a', ['&', 'b', 'c']]

sage: t = propcalc.formula("a & ((~b | c) ^ a -> c) <-> ~b")
sage: t.full_tree()
['<->', ['&', 'a', ['->', ['^', ['|', ['~', 'b'], 'c'], 'a'], 'c']], ['~', 'b
↪']]

sage: f = propcalc.formula("~~(a&~b)")
sage: f.full_tree()
['~', ['~', ['&', 'a', ['~', 'b']]]]
```

Note: This function is used by other functions in the logic module that perform syntactic operations on a boolean formula.

AUTHORS:

- Paul Scurek (2013-08-03)

get_bit (*x*, *c*)

Determine if bit `c` of the number `x` is 1.

INPUT:

- `x` – an integer; this is the number from which to take the bit
- `c` – an integer; this is the but number to be taken, where 0 is the low order bit

OUTPUT:

A boolean to be determined as follows:

- `True` if bit `c` of `x` is 1.
- `False` if bit c of x is not 1.

EXAMPLES:

This example illustrates the use of *get_bit()* :

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b")
sage: s.get_bit(2, 1)
True
sage: s.get_bit(8, 0)
False
```

It is not an error to have a bit out of range:

```
sage: s.get_bit(64, 7)
False
```

Nor is it an error to use a negative number:

```
sage: s.get_bit(-1, 3)
False
sage: s.get_bit(64, -1)
True
sage: s.get_bit(64, -2)
False
```

Note: The 0 bit is the low order bit. Errors should be handled gracefully by a return of `False` , and negative numbers `x` always return `False` while a negative `c` will index from the high order bit.

get_next_op (*str*)

Return the next operator in a string.

INPUT:

- `str` – a string; this contains a logical expression

OUTPUT:

The next operator as a string.

EXAMPLES:

This example illustrates how to find the next operator in a formula:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("f&p")
sage: s.get_next_op("abra|cadabra")
'|'
```

Note: The parameter `str` is not necessarily the string representation of the calling object.

iff (*other*)
Combine two formulas with the `<->` operator.

INPUT:

- `other` – a boolean formula; this is the formula on the right side of the operator

OUTPUT:

A boolean formula of the form `self <-> other`.

EXAMPLES:

This example illustrates how to combine two formulas with '<->':

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b")
sage: f = propcalc.formula("c^d")
sage: s.iff(f)
(a&b)<->(c^d)
```

ifthen (*other*)
Combine two formulas with the `->` operator.

INPUT:

- `other` – a boolean formula; this is the formula on the right side of the operator

OUTPUT:

A boolean formula of the form `self -> other`.

EXAMPLES:

This example illustrates how to combine two formulas with '->':

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b")
sage: f = propcalc.formula("c^d")
sage: s.ifthen(f)
(a&b)->(c^d)
```

implies (*other*)
Determine if calling formula implies other formula.

INPUT:

- `self` – calling object

- other – instance of *BooleanFormula*

OUTPUT:

A boolean value to be determined as follows:

- True - if self implies other
- False - if self does not imply ``other

EXAMPLES:

This example illustrates determining if one formula implies another:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a<->b")
sage: g = propcalc.formula("b->a")
sage: f.implies(g)
True
```

```
sage: h = propcalc.formula("a->(a|~b)")
sage: i = propcalc.formula("a")
sage: h.implies(i)
False
```

AUTHORS:

- Paul Scurek (2013-08-08)

is_contradiction ( )

Determine if the formula is always False .

OUTPUT:

A boolean value to be determined as follows:

- True if the formula is a contradiction.
- False if the formula is not a contradiction.

EXAMPLES:

This example illustrates how to check if a formula is a contradiction.

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a&~a")
sage: f.is_contradiction()
True

sage: f = propcalc.formula("a|~a")
sage: f.is_contradiction()
False

sage: f = propcalc.formula("a|b")
sage: f.is_contradiction()
False
```

is_satisfiable ( )

Determine if the formula is True for some assignment of values.

OUTPUT:

A boolean value to be determined as follows:

- `True` if there is an assignment of values that makes the formula `True` .
- `False` if the formula cannot be made `True` by any assignment of values.

EXAMPLES:

This example illustrates how to check a formula for satisfiability:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a|b")
sage: f.is_satisfiable()
True

sage: g = f & (~f)
sage: g.is_satisfiable()
False
```

is_tautology ( )

Determine if the formula is always `True` .

OUTPUT:

A boolean value to be determined as follows:

- `True` if the formula is a tautology.
- `False` if the formula is not a tautology.

EXAMPLES:

This example illustrates how to check if a formula is a tautology:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a|~a")
sage: f.is_tautology()
True

sage: f = propcalc.formula("a&~a")
sage: f.is_tautology()
False

sage: f = propcalc.formula("a&b")
sage: f.is_tautology()
False
```

polish_notation ( )

Convert the calling boolean formula into polish notation.

OUTPUT:

A string representation of the formula in polish notation.

EXAMPLES:

This example illustrates converting a formula to polish notation:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("~~a|(c->b)")
sage: f.polish_notation()
'|~~a->cb'

sage: g = propcalc.formula("(a|~b)->c")
```

```
sage: g.polish_notation()
'->|a~bc'
```

AUTHORS:

- Paul Scurek (2013-08-03)

reduce_op (*tree*)

Convert if-and-only-if, if-then, and xor operations to operations only involving and/or operations.

INPUT:

- `tree` – a list; this represents a branch of a parse tree

OUTPUT:

A new list with no '^', '->', or '<->' as first element of list.

EXAMPLES:

This example illustrates the use of *reduce_op()* with `apply_func()`:

```
sage: import sage.logic.propcalc as propcalc, sage.logic.logicparser as␣
↪logicparser
sage: s = propcalc.formula("a->b^c")
sage: tree = ['->', 'a', ['^', 'b', 'c']]
sage: logicparser.apply_func(tree, s.reduce_op)
['|', ['~', 'a', None], ['&', ['|', 'b', 'c'], ['~', ['&', 'b', 'c'], None]]]
```

Note: This function only operates on a single branch of a parse tree. To apply the function to an entire parse tree, pass the function as an argument to *apply_func()* in *logicparser*.

satformat ( )

Return the satformat representation of a boolean formula.

OUTPUT:

The satformat of the formula as a string.

EXAMPLES:

This example illustrates how to find the satformat of a formula:

```
sage: import sage.logic.propcalc as propcalc
sage: f = propcalc.formula("a&((b|c)^a->c)<->b")
sage: f.convert_cnf()
sage: f
(a|~b|c)&(a|~b|~c)&(~a|b|~c)
sage: f.satformat()
'p cnf 3 0\n1 -2 3 0 1 -2 -3 \n0 -1 2 -3'
```

Note: See www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps for a description of satformat.

If the instance of boolean formula has not been converted to CNF form by a call to *convert_cnf()* or *convert_cnf_recur()*, then *satformat()* will call *convert_cnf()*. Please see the notes for *convert_cnf()* and *convert_cnf_recur()* for performance issues.

to_infix (*tree*)

Convert a parse tree from prefix to infix form.

INPUT:

- `tree` – a list; this represents a branch of a parse tree

OUTPUT:

A new list.

EXAMPLES:

This example shows how to convert a parse tree from prefix to infix form:

```
sage: import sage.logic.propcalc as propcalc, sage.logic.logicparser as␣
↪logicparser
sage: s = propcalc.formula("(a&b)|(a&c)")
sage: tree = ['|', ['&', 'a', 'b'], ['&', 'a', 'c']]
sage: logicparser.apply_func(tree, s.to_infix)
[['a', '&', 'b'], '|', ['a', '&', 'c']]
```

Note: This function only operates on a single branch of a parse tree. To apply the function to an entire parse tree, pass the function as an argument to *apply_func()* in *logicparser* .

tree ( )

Return the parse tree of this boolean expression.

OUTPUT:

The parse tree as a nested list

EXAMPLES:

This example illustrates how to find the parse tree of a boolean formula:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("man -> monkey & human")
sage: s.tree()
['->', 'man', ['&', 'monkey', 'human']]
```

```
sage: f = propcalc.formula("a & ((~b | c) ^ a -> c) <-> ~b")
sage: f.tree()
['<->',
 ['&', 'a', ['->', ['^', ['|', ['~', 'b', None], 'c'], 'a'], 'c']],
 ['~', 'b', None]]
```

Note: This function is used by other functions in the logic module that perform semantic operations on a boolean formula.

truthtable (*start=0*, *end=-1*)

Return a truth table for the calling formula.

INPUT:

- `start` – (default: 0) an integer; this is the first row of the truth table to be created
- `end` – (default: -1) an integer; this is the laste row of the truth table to be created

OUTPUT:

The truth table as a 2-D array

EXAMPLES:

This example illustrates the creation of a truth table:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b|~(c|a)")
sage: s.truthtable()
a      b      c      value
False  False  False  True
False  False  True   False
False  True   False  True
False  True   True   False
True   False  False  False
True   False  True   False
True   True   False  True
True   True   True   True
```

We can now create a truthtable of rows 1 to 4, inclusive:

```
sage: s.truthtable(1, 5)
a      b      c      value
False  False  True   False
False  True   False  True
False  True   True   False
True   False  False  False
```

Note: Each row of the table corresponds to a binary number, with each variable associated to a column of the number, and taking on a true value if that column has a value of 1. Please see the logictable module for details. The function returns a table that start inclusive and end exclusive so `truthtable(0,2)` will include row 0, but not row 2.

When sent with no start or end parameters, this is an exponential time function requiring $O(2^n)$ time, where n is the number of variables in the expression.

CHAPTER

THREE

EVALUATION OF BOOLEAN FORMULAS

AUTHORS:

- Chris Gorecki (2006): initial version
- Paul Scurek (2013-08-05): updated docstring formatting

EXAMPLES:

We can assign values to the variables and evaluate a formula:

```
sage: import sage.logic.booleval as booleval
sage: t = ['|', ['&', 'a', 'b'], ['&', 'a', 'c']]
sage: d = {'a' : True, 'b' : False, 'c' : True}
sage: booleval.eval_formula(t, d)
True
```

We can change our assignment of values by modifying the dictionary:

```
sage: d['a'] = False
sage: booleval.eval_formula(t, d)
False
```

sage.logic.booleval. **eval_f** (*tree*)

Evaluate the tree.

INPUT:

- `tree` – a list of three elements corresponding to a branch of a parse tree

OUTPUT:

The result of the evaluation as a boolean value.

EXAMPLES:

This example illustrates how to evaluate a parse tree:

```
sage: import sage.logic.booleval as booleval
sage: booleval.eval_f(['&', True, False])
False

sage: booleval.eval_f(['^', True, True])
False

sage: booleval.eval_f(['|', False, True])
True
```

sage.logic.booleval. **eval_formula** (*tree*, *vdict*)

Evaluate the tree and return a boolean value.

INPUT:

- `tree` – a list of three elements corresponding to a branch of a parse tree
- `vdict` – a dictionary containing variable keys and boolean values

OUTPUT:

The result of the evaluation as a boolean value.

EXAMPLES:

This example illustrates evaluating a boolean formula:

```
sage: import sage.logic.booleval as booleval
sage: t = ['|', ['&', 'a', 'b'], ['&', 'a', 'c']]
sage: d = {'a' : True, 'b' : False, 'c' : True}
sage: booleval.eval_formula(t, d)
True
```

```
sage: d['a'] = False
sage: booleval.eval_formula(t, d)
False
```

sage.logic.booleval. **eval_op** (*op*, *lv*, *rv*)

Evaluate `lv` and `rv` according to the operator `op` .

INPUT:

- `op` – a string or character representing a boolean operator
- `lv` – a boolean or variable
- `rv` – a boolean or variable

OUTPUT:

The evaluation of `lv op rv` as a boolean value.

EXAMPLES:

We can evaluate an operator given the values on either side:

```
sage: import sage.logic.booleval as booleval
sage: booleval.eval_op('&', True, False)
False

sage: booleval.eval_op('^', True, True)
False

sage: booleval.eval_op('|', False, True)
True
```

CHAPTER

FOUR

MODULE THAT CREATES AND MODIFIES PARSE TREES OF WELL FORMED BOOLEAN FORMULAS.

A parse tree of a boolean formula is a nested list, where each branch is either a single variable, or a formula composed of either two variables and a binary operator or one variable and a unary operator. The function parse produces a parse tree that is simplified for the purposes of more efficient truth value evaluation. The function *polish_parse()* produces the full parse tree of a boolean formula which is used in functions related to proof and inference. That is, *parse()* is meant to be used with functions in the logic module that perform semantic operations on a boolean formula, and *polish_parse()* is to be used with functions that perform syntactic operations on a boolean formula.

AUTHORS:

- Chris Gorecki (2007): initial version
- Paul Scurek (2013-08-01): added polish_parse, cleaned up python code, updated docstring formatting
- Paul Scurek (2013-08-06): added *recover_formula()*, *recover_formula_internal()*, *prefix_to_infix()*, *to_infix_internal()*
- Paul Scurek (2013-08-08): added get_trees, error handling in polish_parse, recover_formula_internal, and tree_parse

EXAMPLES:

Find the parse tree and variables of a string representation of a boolean formula:

```
sage: import sage.logic.logicparser as logicparser
sage: s = 'a|b&c'
sage: t = logicparser.parse(s)
sage: t
(['|', 'a', ['&', 'b', 'c']], ['a', 'b', 'c'])
```

Find the full syntax parse tree of a string representation of a boolean formula:

```
sage: import sage.logic.logicparser as logicparser
sage: s = '(a&b)->~~c'
sage: logicparser.polish_parse(s)
['->', ['&', 'a', 'b'], ['~', ['~', 'c']]]
```

Find the tokens and distinct variables of a boolean formula:

```
sage: import sage.logic.logicparser as logicparser
sage: s = '~(a|~b)<->(c->c)'
sage: logicparser.tokenize(s)
(['(', '~', '(', 'a', '|', '~', 'b', ')', '<->', '(', 'c', '->', 'c', ')', ')'], ['a
↪', 'b', 'c'])
```

Find the parse tree of a boolean formula from a list of the formula's tokens:

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['(', 'a', '->', '~', 'c', ')']
sage: logicparser.tree_parse(t)
['->', 'a', ['~', 'c', None]]
sage: r = ['(', '~', '~', 'a', '|', 'b', ')']
sage: logicparser.tree_parse(r)
['|', 'a', 'b']
```

Find the full syntax parse tree of a boolean formula from a list of tokens:

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['(', 'a', '->', '~', 'c', ')']
sage: logicparser.tree_parse(t, polish = True)
['->', 'a', ['~', 'c']]
sage: r = ['(', '~', '~', 'a', '|', 'b', ')']
sage: logicparser.tree_parse(r, polish = True)
['|', ['~', ['~', 'a']], 'b']
```

sage.logic.logicparser. **apply_func** (*tree*, *func*)

Apply `func` to each node of `tree` , and return a new parse tree.

INPUT:

- `tree` – a parse tree of a boolean formula
- `func` – a function to be applied to each node of tree; this may be a function that comes from elsewhere in the logic module

OUTPUT:

The new parse tree in the form of a nested list.

EXAMPLES:

This example uses *`apply_func()`* where `func` switches two entries of tree:

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['|', ['&', 'a', 'b'], ['&', 'a', 'c']]
sage: f = lambda t: [t[0], t[2], t[1]]
sage: logicparser.apply_func(t, f)
['|', ['&', 'c', 'a'], ['&', 'b', 'a']]
```

sage.logic.logicparser. **get_trees** (**statements*)

Return the full syntax parse trees of the statements.

INPUT:

- `*statements` – strings or `BooleanFormula` instances

OUTPUT:

The parse trees in a list.

EXAMPLES:

This example illustrates finding the parse trees of multiple formulas.

```
sage: import sage.logic.propcalc as propcalc
sage: import sage.logic.logicparser as logicparser
sage: f = propcalc.formula("((a|b)&~~c)")
sage: g = "a<->(~(c))"
```

```
sage: h = "~b"
sage: logicparser.get_trees(f, g, h)
[['&', ['|', 'a', 'b'], ['~', ['~', 'c']]],
['<->', 'a', ['~', 'c']],
['~', 'b']]
```

```
sage: i = "(~q->p)"
sage: j = propcalc.formula("a")
sage: logicparser.get_trees(i, j)
[['->', ['~', 'q'], 'p'], ['a']]
```

```
sage: k = "p"
sage: logicparser.get_trees(k)
[['p']]
```

AUTHORS:

- Paul Scurek (2013-08-06)

sage.logic.logicparser.**parse**(*s*)

Return a parse tree from a boolean formula `s`.

INPUT:

- `s` – a string containing a boolean formula

OUTPUT:

A list containing the prase tree and a list containing the variables in a boolean formula in this order:

1. the list containing the pase tree
2. the list containing the variables

EXAMPLES:

This example illustrates how to produce the parse tree of a boolean formula `s`:

```
sage: import sage.logic.logicparser as logicparser
sage: s = 'a|b&c'
sage: t = logicparser.parse(s)
sage: t
(['|', 'a', ['&', 'b', 'c']], ['a', 'b', 'c'])
```

sage.logic.logicparser.**parse_ltor**(*toks*, *n=0*, *polish=False*)

Return a parse tree from `toks`, where each token in `toks` is atomic.

INPUT:

- `toks` – a list of tokens. Each token is atomic.
- `n` – (default: 0) an integer representing which order of operations are occurring
- `polish` – (default: `False`) a boolean; when `True`, double negations are not cancelled and negated statements are turned into list of length two.

OUTPUT:

The parse tree as a nested list that depends on `polish` as follows:

- If `False`, then return a simplified parse tree.
- If `True`, then return the full syntax parse tree.

EXAMPLES:

This example illustrates the use of *parse_ltor()* when `polish` is `False`:

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['a', '|', 'b', '&', 'c']
sage: logicparser.parse_ltor(t)
['|', 'a', ['&', 'b', 'c']]
```

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['a', '->', '~', '~', 'b']
sage: logicparser.parse_ltor(t)
['->', 'a', 'b']
```

We now repeat the previous example, but with `polish` being `True`:

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['a', '->', '~', '~', 'b']
sage: logicparser.parse_ltor(t, polish = True)
['->', 'a', ['~', ['~', 'b']]]
```

sage.logic.logicparser.**polish_parse**(*s*)

Return the full syntax parse tree from a boolean formula `s`.

INPUT:

- `s` – a string containing a boolean expression

OUTPUT:

The full syntax parse tree as a nested list.

EXAMPLES:

This example illustrates how to find the full syntax parse tree of a boolean formula:

```
sage: import sage.logic.logicparser as logicparser
sage: s = 'a|~~b'
sage: t = logicparser.polish_parse(s)
sage: t
['|', 'a', ['~', ['~', 'b']]]
```

AUTHORS:

- Paul Scurek (2013-08-03)

sage.logic.logicparser.**prefix_to_infix**(*prefix_tree*)

Convert a parse tree from prefix form to infix form.

INPUT:

- `prefix_tree` – a list; this is a full syntax parse tree in prefix form

OUTPUT:

A list containing the tree in infix form.

EXAMPLES:

This example illustrates converting a prefix tree to an infix tree:

```
sage: import sage.logic.logicparser as logicparser
sage: import sage.logic.propcalc as propcalc
sage: t = ['|', ['~', 'a'], ['&', 'b', 'c']]
sage: logicparser.prefix_to_infix(t)
[['~', 'a'], '|', ['b', '&', 'c']]
```

```
sage: f = propcalc.formula("(a&~b)<->~~~(c|d)")
sage: logicparser.prefix_to_infix(f.full_tree())
[['a', '&', ['~', 'b']], '<->', ['~', ['~', ['~', ['c', '|', 'd']]]]]
```

Note: The function *polish_parse()* may be passed as an argument, but *tree_parse()* may not unless the parameter `polish` is set to `True` .

AUTHORS:

- Paul Scurek (2013-08-06)

`sage.logic.logicparser.` **`recover_formula`** (*prefix_tree*)

Recover the formula from a parse tree in prefix form.

INPUT:

- `prefix_tree` – a list; this is a full syntax parse tree in prefix form

OUTPUT:

The formula as a string.

EXAMPLES:

This example illustrates the recovery of a formula from a parse tree:

```
sage: import sage.logic.propcalc as propcalc
sage: import sage.logic.logicparser as logicparser
sage: t = ['->', ['&', 'a', ['~', ['~', 'c']]], ['~', ['|', ['~', 'c'], 'd']]]
sage: logicparser.recover_formula(t)
'(a&~~c)->~(~c|d)'

sage: f = propcalc.formula("a&(~~c|d)")
sage: logicparser.recover_formula(f.full_tree())
'a&(~~c|d)'

sage: r = ['~', 'a']
sage: logicparser.recover_formula(r)
'~a'

sage: s = ['d']
sage: logicparser.recover_formula(s)
'd'
```

Note: The function *polish_parse()* may be passed as an argument, but *tree_parse()* may not unless the parameter `polish` is set to `True` .

AUTHORS:

- Paul Scurek (2013-08-06)

sage.logic.logicparser. **recover_formula_internal** (*prefix_tree*)
Recover the formula from a parse tree in prefix form.

INPUT:

- prefix_tree – a list; this is a simple tree with at most one operator in prefix form

OUTPUT:

The formula as a string.

EXAMPLES:

This example illustrates recovering the formula from a parse tree:

```
sage: import sage.logic.logicparser as logicparser
sage: import sage.logic.propcalc as propcalc
sage: t = ['->', 'a', 'b']
sage: logicparser.recover_formula_internal(t)
'(a->b)'

sage: r = ['~', 'c']
sage: logicparser.recover_formula_internal(r)
'~c'

sage: s = ['d']
sage: logicparser.recover_formula_internal(s)
'd'
```

We can pass *recover_formula_internal()* as an argument in *apply_func()* :

```
sage: f = propcalc.formula("~(d|c)<->(a&~~~c)")
sage: logicparser.apply_func(f.full_tree(), logicparser.recover_formula_internal)
'(~(d|c)<->(a&~~~c))'
```

Note: This function is for internal use by *logicparser* . The function recovers the formula of a simple parse tree in prefix form. A simple parse tree contains at most one operator.

The function *polish_parse()* may be passed as an argument, but *tree_parse()* may not unless the parameter polish is set to True .

AUTHORS:

- Paul Scurek (2013-08-06)

sage.logic.logicparser. **to_infix_internal** (*prefix_tree*)
Convert a simple parse tree from prefix form to infix form.

INPUT:

- prefix_tree – a list; this is a simple parse tree in prefix form with at most one operator

OUTPUT:

The tree in infix form as a list.

EXAMPLES:

This example illustrates converting a simple tree from prefix to infix form:

```
sage: import sage.logic.logicparser as logicparser
sage: import sage.logic.propcalc as propcalc
sage: t = ['|', 'a', 'b']
sage: logicparser.to_infix_internal(t)
['a', '|', 'b']
```

We can pass *to_infix_internal()* as an argument in *apply_func()* :

```
sage: f = propcalc.formula("(a&~b)<->~~~(c|d)")
sage: logicparser.apply_func(f.full_tree(), logicparser.to_infix_internal)
[['a', '&', ['~', 'b']], '<->', ['~', ['~', ['~', ['c', '|', 'd']]]]]
```

Note: This function is for internal use by *logicparser* . It converts a simple parse tree from prefix form to infix form. A simple parse tree contains at most one operator.

The function *polish_parse()* may be passed as an argument, but *tree_parse()* may not unless the parameter `polish` is set to `True` .

AUTHORS:

- Paul Scurek (2013-08-06)

`sage.logic.logicparser.` **`tokenize`** (*s*)

Return the tokens and the distinct variables appearing in a boolean formula `s` .

INPUT:

- `s` – a string representation of a boolean formula

OUTPUT:

The tokens and variables as an ordered pair of lists in the following order:

1. A list containing the tokens of `s` , in the order they appear in `s`
2. A list containing the distinct variables in `s` , in the order they appear in `s`

EXAMPLES:

This example illustrates how to tokenize a string representation of a boolean formula:

```
sage: import sage.logic.logicparser as logicparser
sage: s = 'a|b&c'
sage: t = logicparser.tokenize(s)
sage: t
(['(', 'a', '|', 'b', '&', 'c', ')'], ['a', 'b', 'c'])
```

`sage.logic.logicparser.` **`tree_parse`** (*toks*, *polish=False*)

Return a parse tree from the tokens in `toks` .

INPUT:

- `toks` – a list of tokens from a boolean formula
- `polish` – (default: `False`) a boolean; when `True` , *tree_parse()* will return the full syntax parse tree

OUTPUT:

A parse tree in the form of a nested list that depends on `polish` as follows:

- If `False` , then return a simplified parse tree.

- If `True` , then return the full syntax parse tree.

EXAMPLES:

This example illustrates the use of *tree_parse()* when `polish` is `False` :

```
sage: import sage.logic.logicparser as logicparser
sage: t = ['(', 'a', '|', 'b', '&', 'c', ')']
sage: logicparser.tree_parse(t)
['|', 'a', ['&', 'b', 'c']]
```

We now demonstrate the use of *tree_parse()* when `polish` is `True` :

```
sage: t = ['(', 'a', '->', '~', '~', 'b', ')']
sage: logicparser.tree_parse(t)
['->', 'a', 'b']
sage: t = ['(', 'a', '->', '~', '~', 'b', ')']
sage: logicparser.tree_parse(t, polish = True)
['->', 'a', ['~', ['~', 'b']]]
```

CHAPTER

FIVE

SYMBOLIC LOGIC EXPRESSIONS

An expression is created from a string that consists of the operators `!` , `&` , `|` , `->` , `<->` , which correspond to the logical functions not, and, or, if then, if and only if, respectively. Variable names must start with a letter and contain only alpha-numerics and the underscore character.

AUTHORS:

- Chris Gorecki (2007): initial version
- William Stein (2007-08-31): integration into Sage 2.8.4
- Paul Scurek (2013-08-03): updated docstring formatting

class sage.logic.logic. **SymbolicLogic**

EXAMPLES:

This example illustrates how to create a boolean formula and print its table:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: t = log.truthtable(s)
sage: log.print_table(t)
a     | b     | c     | value |
-------------------------------
False | False | False | True  |
False | False | True  | False |
False | True  | False | True  |
False | True  | True  | False |
True  | False | False | False |
True  | False | True  | False |
True  | True  | False | True  |
True  | True  | True  | True  |
```

combine (*statement1*, *statement2*)

Return a new statement which contains the two statements or'd together.

INPUT:

- `statement1` – the first statement
- `statement2` – the second statement

OUTPUT:

A new staement which or'd the given statements together.

EXAMPLES:

```
sage: log = SymbolicLogic()
sage: s1 = log.statement("(a&b)")
sage: s2 = log.statement("b")
sage: log.combine(s1,s2)
[['OPAREN',
  'OPAREN',
  'OPAREN',
  'a',
  'AND',
  'b',
  'CPAREN',
  'CPAREN',
  'OR',
  'OPAREN',
  'b',
  'CPAREN',
  'CPAREN'],
 {'a': 'False', 'b': 'False'},
 ['a', 'b', 'b']]
```

print_table (*table*)

Return a truthtable corresponding to the given statement.

INPUT:

- `table` – object created by *truthtable()* method; it contains the variable values and the evaluation of the statement

OUTPUT:

A formatted version of the truth table.

EXAMPLES:

This example illustrates the creation of a statement and its truth table:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: t = log.truthtable(s) #creates the whole truth table
sage: log.print_table(t)
a     | b     | c     | value |
--------------------------------
False | False | False | True  |
False | False | True  | False |
False | True  | False | True  |
False | True  | True  | False |
True  | False | False | False |
True  | False | True  | False |
True  | True  | False | True  |
True  | True  | True  | True  |
```

We can also print a shortened table:

```
sage: t = log.truthtable(s, 1, 5)
sage: log.print_table(t)
a     | b     | c     | value | value |
----------------------------------------
False | False | False | True  | True  |
False | False | True  | False | False |
```

```
False | False | True  | True  | False |
False | True  | False | False | True  |
```

prove (*statement*)

A function to test to see if the statement is a tautology or contradiction by calling a C++ library.

Todo

Implement this method.

EXAMPLES:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: log.prove(s)
Traceback (most recent call last):
...
NotImplementedError
```

simplify (*table*)

Call a C++ implementation of the ESPRESSO algorithm to simplify the given truth table.

Todo

Implement this method.

EXAMPLES:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: t = log.truthtable(s)
sage: log.simplify(t)
Traceback (most recent call last):
...
NotImplementedError
```

statement (*s*)

Return a token list to be used by other functions in the class

INPUT:

- `s` – a string containing the logic expression to be manipulated
- `global vars` – a dictionary with variable names as keys and the variables' current boolean values as dictionary values
- `global vars_order` – a list of the variables in the order that they are found

OUTPUT:

A list of length three containing the following in this order:

1. a list of tokens
2. a dictionary of variable/value pairs
3. a list of the variables in the order they were found

EXAMPLES:

This example illustrates the creation of a statement:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: s2 = log.statement("!((!(a&b)))")
```

It is an error to use invalid variable names:

```
sage: s = log.statement("3fe & @q")
Invalid variable name:  3fe
Invalid variable name:  @q
```

It is also an error to use invalid syntax:

```
sage: s = log.statement("a&&b")
Malformed Statement
sage: s = log.statement("a&((b)")
Malformed Statement
```

truthtable (*statement*, *start=0*, *end=-1*)

Return a truth table.

INPUT:

- `statement` – a list; it contains the tokens and the two global variables vars and vars_order
- `start` – (default: 0) an integer; this represents the row of the truth table from which to start
- `end` – (default: -1) an integer; this represents the last row of the truth table to be created

OUTPUT:

The truth table as a 2d array with the creating formula tacked to the front.

EXAMPLES:

This example illustrates the creation of a statement:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: t = log.truthtable(s) #creates the whole truth table
```

We can now create truthtable of rows 1 to 5:

```
sage: s2 = log.truthtable(s, 1, 5); s2
[[['OPAREN',
   'a',
   'AND',
   'b',
   'OR',
   'NOT',
   'OPAREN',
   'c',
   'OR',
   'a',
   'CPAREN',
   'CPAREN'],
  {'a': 'False', 'b': 'False', 'c': 'True'},
  ['a', 'b', 'c']],
 ['False', 'False', 'True', 'False'],
```

```
['False', 'True', 'False', 'True'],
['False', 'True', 'True', 'True'],
['True', 'False', 'False', 'False']]
```

Note: When sent with no start or end parameters this is an exponential time function requiring $O(2^n)$ time, where n is the number of variables in the logic expression

sage.logic.logic. **eval** (*toks*)

Evaluate the expression contained in `toks` .

INPUT:

- `toks` – a list of tokens; this represents a boolean expression

OUTPUT:

A boolean value to be determined as follows:

- `True` if expression evaluates to `True` .
- `False` if expression evaluates to `False` .

Note: This function is for internal use by the *SymbolicLogic* class. The evaluations rely on setting the values of the variables in the global dictionary vars.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!(c|a)")
sage: sage.logic.logic.eval(s[0])
'True'
```

sage.logic.logic. **eval_and_op** (*lval*, *rval*)

Apply the 'and' operator to `lval` and `rval` .

INPUT:

- `lval` – a string; this represents the value of the variable appearing to the left of the 'and' operator
- `rval` – a string; this represents the value of the variable appearing to the right of the 'and' operator

OUTPUT:

The result of applying 'and' to `lval` and `rval` as a string.

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: sage.logic.logic.eval_and_op('False', 'False')
'False'
sage: sage.logic.logic.eval_and_op('False', 'True')
'False'
sage: sage.logic.logic.eval_and_op('True', 'False')
'False'
sage: sage.logic.logic.eval_and_op('True', 'True')
'True'
```

sage.logic.logic. **eval_bin_op** (*args*)

Return a boolean value based on the truth table of the operator in `args` .

INPUT:

- `args` – a list of length 3; this contains a variable name, then a binary operator, and then a variable name, in that order

OUTPUT:

A boolean value; this is the evaluation of the operator based on the truth values of the variables.

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("!(a&b)"); s
[['OPAREN', 'NOT', 'OPAREN', 'a', 'AND', 'b', 'CPAREN', 'CPAREN'],
 {'a': 'False', 'b': 'False'},
 ['a', 'b']]
sage: sage.logic.logic.eval_bin_op(['a', 'AND', 'b'])
'False'
```

sage.logic.logic. **eval_iff_op** (*lval*, *rval*)

Apply the 'if and only if' operator to `lval` and `rval` .

INPUT:

- `lval` – a string; this represents the value of the variable appearing to the left of the 'if and only if' operator
- `rval` – a string; this represents the value of the variable appearing to the right of the 'if and only if' operator

OUTPUT:

A string representing the result of applying 'if and only if' to `lval` and `rval` .

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: sage.logic.logic.eval_iff_op('False', 'False')
'True'
sage: sage.logic.logic.eval_iff_op('False', 'True')
'False'
sage: sage.logic.logic.eval_iff_op('True', 'False')
'False'
sage: sage.logic.logic.eval_iff_op('True', 'True')
'True'
```

sage.logic.logic. **eval_ifthen_op** (*lval*, *rval*)

Apply the 'if then' operator to `lval` and `rval` .

INPUT:

- `lval` – a string; this represents the value of the variable appearing to the left of the 'if then' operator
- `rval` – a string;t his represents the value of the variable appearing to the right of the 'if then' operator

OUTPUT:

A string representing the result of applying 'if then' to `lval` and `rval` .

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: sage.logic.logic.eval_ifthen_op('False', 'False')
'True'
sage: sage.logic.logic.eval_ifthen_op('False', 'True')
'True'
sage: sage.logic.logic.eval_ifthen_op('True', 'False')
'False'
sage: sage.logic.logic.eval_ifthen_op('True', 'True')
'True'
```

sage.logic.logic. **eval_ltor_toks** (*lrtoks*)

Evaluates the expression contained in `lrtoks` .

INPUT:

- `lrtoks` – a list of tokens; this represents a part of a boolean formula that contains no inner parentheses

OUTPUT:

A boolean value to be determined as follows:

- `True` if expression evaluates to `True` .
- `False` if expression evaluates to `False` .

Note: This function is for internal use by the *SymbolicLogic* class. The evaluations rely on setting the values of the variables in the global dictionary vars.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|!c")
sage: ltor = s[0][1:-1]; ltor
['a', 'AND', 'b', 'OR', 'NOT', 'c']
sage: sage.logic.logic.eval_ltor_toks(ltor)
'True'
```

sage.logic.logic. **eval_mon_op** (*args*)

Return a boolean value based on the truth table of the operator in `args` .

INPUT:

- `args` – a list of length 2; this contains the token 'NOT' and then a variable name

OUTPUT:

A boolean value to be determined as follows:

- `True` if the variable in `args` is `False` .
- `False` if the variable in `args` is `True` .

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("!(a&b)|!a"); s
[['OPAREN', 'NOT', 'OPAREN', 'a', 'AND', 'b', 'CPAREN', 'OR', 'NOT', 'a', 'CPAREN
↪'],
 {'a': 'False', 'b': 'False'},
 ['a', 'b']]
sage: sage.logic.logic.eval_mon_op(['NOT', 'a'])
'True'
```

sage.logic.logic. **eval_or_op** (*lval*, *rval*)

Apply the 'or' operator to `lval` and `rval` .

INPUT:

- `lval` – a string; this represents the value of the variable appearing to the left of the 'or' operator
- `rval` – a string; this represents the value of the variable appearing to the right of the 'or' operator

OUTPUT:

A string representing the result of applying 'or' to `lval` and `rval` .

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: sage.logic.logic.eval_or_op('False', 'False')
'False'
sage: sage.logic.logic.eval_or_op('False', 'True')
'True'
sage: sage.logic.logic.eval_or_op('True', 'False')
'True'
sage: sage.logic.logic.eval_or_op('True', 'True')
'True'
```

sage.logic.logic. **get_bit** (*x*, *c*)

Determine if bit `c` of the number `x` is 1.

INPUT:

- `x` – an integer; this is the number from which to take the bit
- `c` – an integer; this is the bit number to be taken

OUTPUT:

A boolean value to be determined as follows:

- `True` if bit `c` of `x` is 1.
- `False` if bit `c` of `x` is not 1.

Note: This function is for internal use by the *SymbolicLogic* class.

EXAMPLES:

```
sage: from sage.logic.logic import get_bit
sage: get_bit(int(2), int(1))
'True'
sage: get_bit(int(8), int(0))
'False'
```

sage.logic.logic. **reduce_bins** (*lrtoks*)

Evaluate `lrtoks` to a single boolean value.

INPUT:

- `lrtoks` – a list of tokens; this represents a part of a boolean formula that contains no inner parentheses or monotonic operators

OUTPUT:

`None` ; the pointer to lrtoks is now a list containing `True` or `False` .

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("a&b|c")
sage: lrtoks = s[0][1:-1]; lrtoks
['a', 'AND', 'b', 'OR', 'c']
sage: sage.logic.logic.reduce_bins(lrtoks); lrtoks
['False']
```

sage.logic.logic. **reduce_monos** (*lrtoks*)

Replace monotonic operator/variable pairs with a boolean value.

INPUT:

- `lrtoks` – a list of tokens; this represents a part of a boolean expression that contains now inner parentheses

OUTPUT:

`None` ; the pointer to `lrtoks` is now a list containing monotonic operators.

Note: This function is for internal use by the *SymbolicLogic* class.

TESTS:

```
sage: log = SymbolicLogic()
sage: s = log.statement("!a&!b")
sage: lrtoks = s[0][1:-1]; lrtoks
['NOT', 'a', 'AND', 'NOT', 'b']
sage: sage.logic.logic.reduce_monos(lrtoks); lrtoks
['True', 'AND', 'True']
```

sage.logic.logic. **tokenize** (*s*, *toks*)

Tokenize `s` and place the tokens of `s` in `toks` .

INPUT:

- `s` – a string; this contains a boolean expression
- `toks` – a list; this will be populated with the tokens of `s`

OUTPUT:

`None` ; the tokens of `s` are placed in `toks` .

Note: This function is for internal use by the *`SymbolicLogic`* class.

EXAMPLES:

```
sage: from sage.logic.logic import tokenize
sage: toks = []
sage: tokenize("(a&b)|c", toks)
sage: toks
['OPAREN', 'a', 'AND', 'b', 'CPAREN', 'OR', 'c', 'CPAREN']
```

CHAPTER

SIX

LOGIC TABLES

A logic table is essentially a 2-D array that is created by the statement class and stored in the private global variable table, along with a list containing the variable names to be used, in order.

The order in which the table is listed essentially amounts to counting in binary. For instance, with the variables A, B, and C the truth table looks like:

```
A      B      C      value
False  False  False    ?
False  False  True     ?
False  True   False    ?
False  True   True     ?
True   False  False    ?
True   False  True     ?
True   True   False    ?
True   True   True     ?
```

This is equivalent to counting in binary, where a table would appear thus;

```
2^2 2^1 2^0 value
0   0   0     0
0   0   1     1
0   1   0     2
0   1   1     3
1   0   0     4
1   0   1     5
1   1   0     6
1   1   1     7
```

Given that a table can be created corresponding to any range of acceptable values for a given statement, it is easy to find the value of a statement for arbitrary values of its variables.

AUTHORS:

- William Stein (2006): initial version
- Chris Gorecki (2006): initial version
- Paul Scurek (2013-08-03): updated docstring formatting

EXAMPLES:

Create a truth table of a boolean formula:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("a&b|~(c|a)")
sage: s.truthtable()
a      b      c      value
```

```
False  False  False  True
False  False  True   False
False  True   False  True
False  True   True   False
True   False  False  False
True   False  True   False
True   True   False  True
True   True   True   True
```

Get the letex code for a truth table:

```
sage: latex(s.truthtable(5,11))
\\\begin{tabular}{llll}c & b & a & value \\\hline True & False & True & False \\True &
↪ True & False & True \\True & True & True & True\end{tabular}
```

It is not an error to use nonsensical numeric inputs:

```
sage: s = propcalc.formula("a&b|~(c|a)")
sage: s.truthtable(5, 9)
a      b      c      value
True   False  True   False
True   True   False  True
True   True   True   True

sage: s.truthtable(9, 5)
a      b      c      value
```

If one argument is provided, truthtable defaults to the end:

```
sage: s.truthtable(-1)
a      b      c      value
False  False  False  True
False  False  True   False
False  True   False  True
False  True   True   False
True   False  False  False
True   False  True   False
True   True   False  True
True   True   True   True
```

If the second argument is negative, truthtable defaults to the end:

```
sage: s.truthtable(4, -2)
a      b      c      value
True   False  False  False
True   False  True   False
True   True   False  True
True   True   True   True
```

Note: For statements that contain a variable list that when printed is longer than the latex page, the columns of the table will run off the screen.

---

class sage.logic.logictable. **Truthtable** (*t*, *vo*)

A truth table.

INPUT:

- `t` – a 2-D array containing the table values
- `vo` – a list of the variables in the expression in order, with each variable occurring only once

get_table_list ( )

Return a list representation of the calling table object.

OUTPUT:

A list representation of the table.

EXAMPLES:

This example illustrates how to show the table as a list:

```
sage: import sage.logic.propcalc as propcalc
sage: s = propcalc.formula("man->monkey&human")
sage: s.truthtable().get_table_list()
 [['man', 'monkey', 'human'], [False, False, False, True], [False, False,␣
↪True, True], [False, True, False, True], [False, True, True, True], [True,␣
↪False, False, False], [True, False, True, False], [True, True, False,␣
↪False], [True, True, True, True]]
```

CHAPTER

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

I

A

B

C

D

E

F

G

I

P

R

S

T

V