cp-logic

Some tools for classical propositional logic - my bachelor thesis.

Prerequisites

- Python 2.7
- Pyside, the QT library for Python
- Pygame

Note: For Ubuntu, simply enter

```
sudo add-apt-repository ppa:pyside
sudo apt-get update
sudo apt-get install python-pyside
sudo apt-get install python-pygame
```

to install Pyside and Pygame. For other OS see the link above (binary libraries for Windows, MAC and Linux are provided).

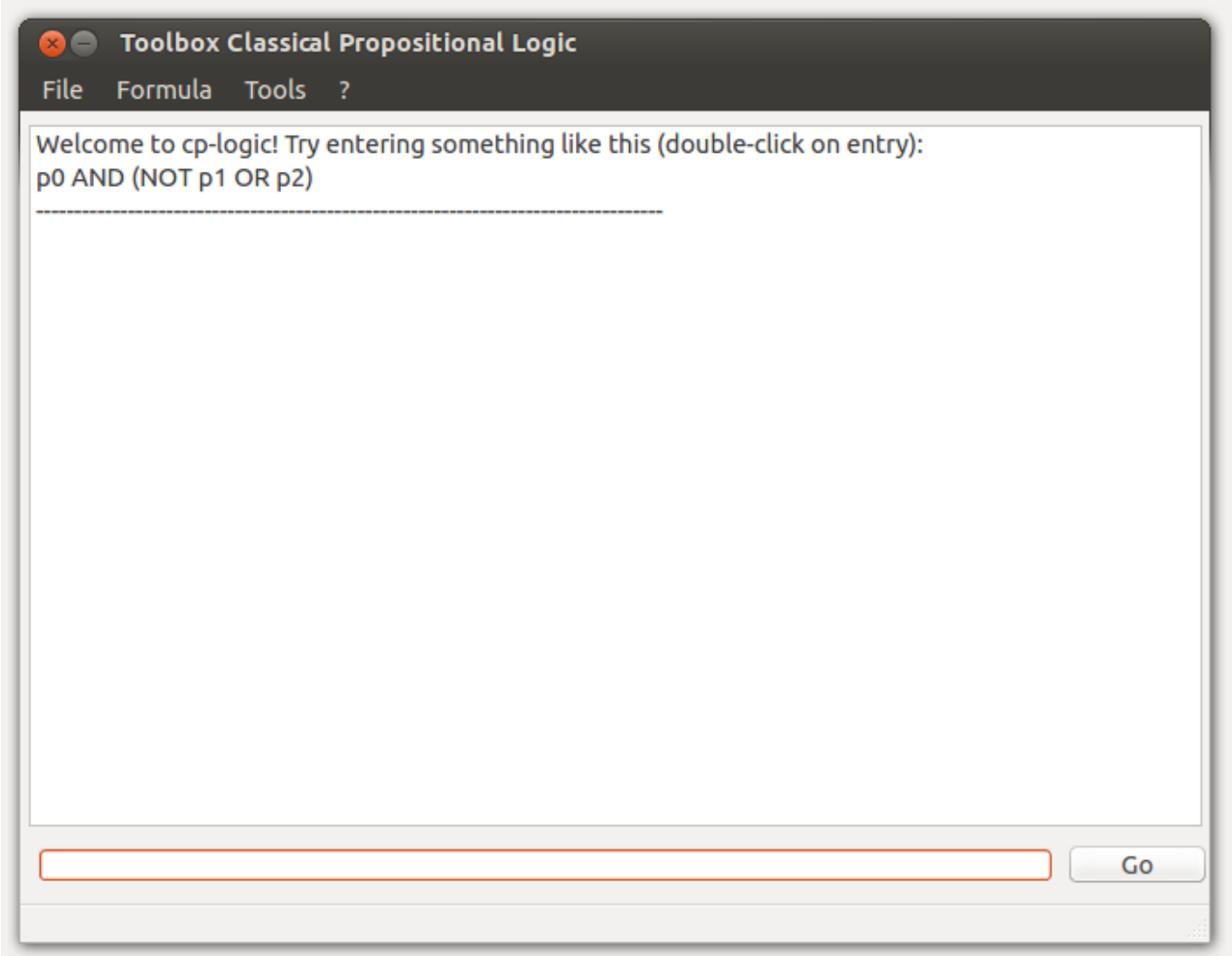
Getting started

To get started, just enter

```
python main.py
```

in the directory containing the Python code. If you're using multiple versions of Python, you may specify the version by entering `python2.7 main.py`.

You should then see a window (depending on your operating system):



Examples

The tokens you can use are listed in the following table. You can enter them as plain ASCII charcters or as Unicode symbols (via copy-paste).

ASCII character	unicode symbol	meaning
`p0`	p_0	A basic proposition. Must contain an index number.
`AND`	∧	logical AND
`OR`	v	logical OR
`NOT`	¬	logical NOT
`IMPL`	→	implication
`TOP`	T	TOP (always evaluated as true)
`BOTTOM`	⊥	BOTTOM (always evaluated as false)
`( )`	( )	brackets, can be used to structure the formula
`A`	A	capital letters always stand for metavariables

Defining a formula

You can either just enter a formula directly

```
p0 AND p1
p3 OR NOT (p1 AND TOP)
```

or you can make an assignment to a so-called metavariable. You can use the capital letters A-Z for storing metavariables.

```
A = NOT p0 AND (p1 AND p2 IMPL NOT p1)
B = p3 OR NOT (p2 AND p4)
```

Metavariables can be used to define other formulas or metavariables:

```
C = A AND NOT B
```

Application of a function

Several forms of a formula can be calculated using the corresponding functions. For example, the *negation normal form** form can be obtained with

```
nnf(p0 AND p1 IMPL p2)
```

which will result in `(NOT p0 OR NOT p1) OR p2`. You can also assign it directly to another variable:

```
D = nnf(A OR p2)
```

For further investigation, it's important to have a binary tree of a given formula. There's an intern representation of the formulat called `pedantic`, which you can also call explicitly.

```
C = pedantic(B)
```

You can get the **conjunctive normal form** by entering

```
cnf(D)
```

You can use functions like nnf, cnf or pedantic directly in the middle of a formula, they will be evaluated first:

```
C = cnf(A) OR NOT nnf(NOT B OR p0)
```

There are many other functions which (instead of a formula) will give you an explicit result like a boolean value, an integer or a drawn tree. For a simple satisfiability test, use

```
sat(A)
```

This function will return a satisfying assignment (if any) for A.

In order to obtain the clause set of a formula A enter

```
clause_set(A)
```

which will provide the corresponding clause set, obtained by the cnf of A. For the length, enter

```
length(A)
```

This function will return the number of occurrences of atomic formulas and logical connectives in A.

Formulas can also be divided and subdivided in their subformulas. For a given formula, all subformulas are shown using

```
sufo(B)
```

Note that here and also with other functions, the result of the function strongly depends on the form of the formula. For example

```
sufo(p0 IMPL p1)
```

and

```
sufo(nnf(p0 IMPL p1))
```

will result in a different set of subformulas.

You can find the truth value which is equivalent to a formula without propositional variables as it is shown in the following example:

```
evaluate(TOP AND ( BOTTOM OR BOTTOM ))
```

For a representation that you can copy-paste into LaTeX, use the `latex`-command:

```
latex(C)
```

Resolution

As seen,

```
clause_set(A)
```

will return all clauses corresponding to the formula A. You can also apply resolution directly:

```
resolution(A)
```

The program will then, according to the rules of resolution, try to generate the empty set. The first 10 clauses will be shown. The program will report

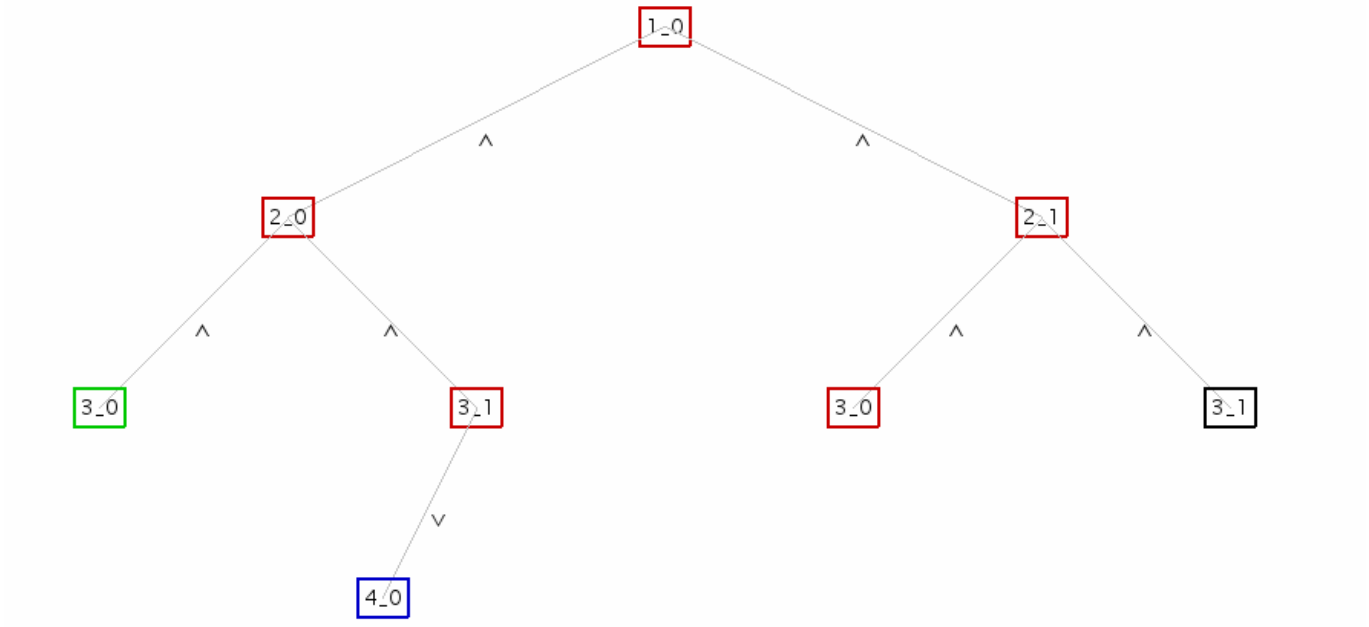
wether the empty clause was found (which means that the formula is not satisfiable) or not (which means that the formula is satisfiable).

Deduction chains

For building up series of deduction chains (dchains for short), you can use the command

```
dchains(A, B, C, ...)
```

where A, B and C are metavariables or formulas, entered directly. The system will provide a graphical output similar to this one:



By clicking on a node, you can see which sequence the node is representing. The color of each node represents whether the underlying sequence is an axiom or not:

- **black**: sequence is reducible
- **red**: sequence is irreducible and not an axiom
- **green**: sequence is an axiom (*true* or *identity*) of PSC

Nesting

Functions are nestable to the first degree:

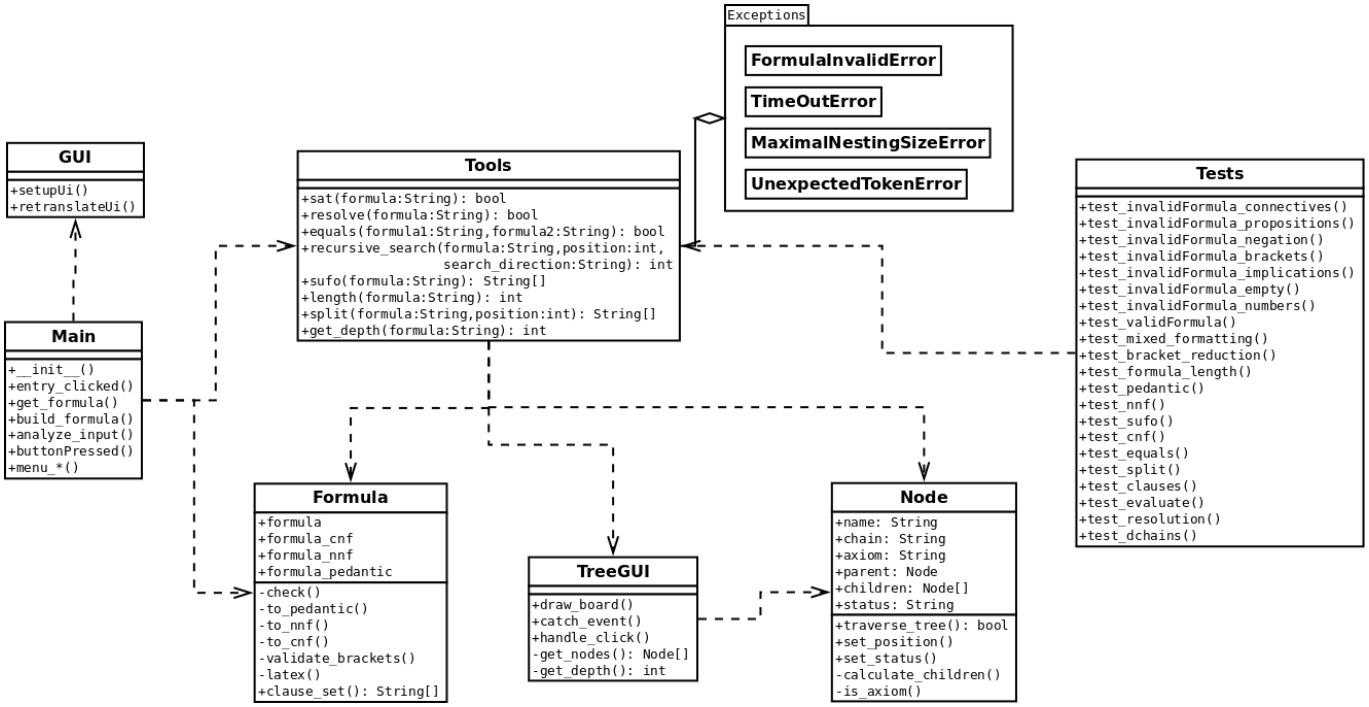
```
length(nnf(A))
sufo(cnf(A))
latex(pedantic(A))
```

Documentation

Available Functions

function	meaning
`length()`	Returns the length of the given formula
`sufo()`	Returns all subformulas from the given formula
`latex()`	Returns a latex representation of the given formula
`pedantic()`	Returns a pedantic (binary) representation of the given formula
`nnf()`	Returns the given formula converted in negation normal form
`cnf()`	Returns the given formula converted to conjunctive normal form
`sat()`	Returns, if possible, a valid valuation for the given formula
`clause_set()`	Calculates the corresponding clause set for the given formula
`evaluate()`	Evaluate the formula (without propositions)
`resolution()`	Apply resolution
`dchains()`	Use D-chains to prove formula

UML



Tests

To run the provided tests, enter

```
python tests.py
```