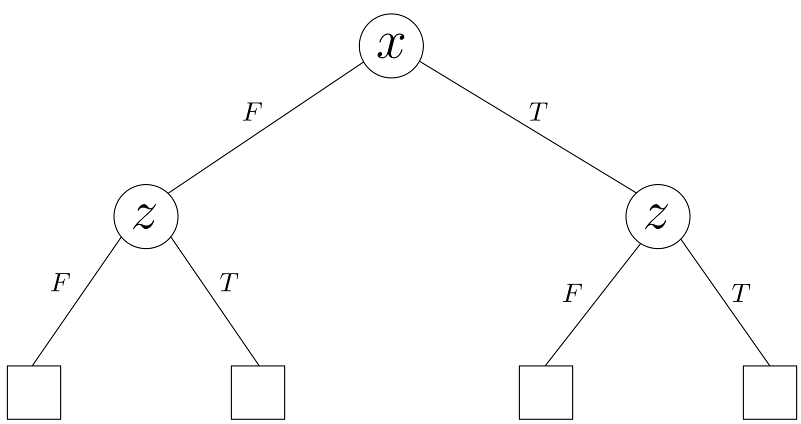
# CS 181U Applied Logic HW 2
# Due Thursday Feb 20, 2020

Your Name Goes Here

**Problem 1: Binary Decision Diagrams.** The following three parts are about binary decision diagrams (BDDs).

**Part I.** Consider the propositional formula $f \equiv x \vee z$.

a. Fill in the terminal nodes for the ordered binary decision diagram for $f$ using the variable ordering $x > z$. (You may find it helpful to write out the truth table.)

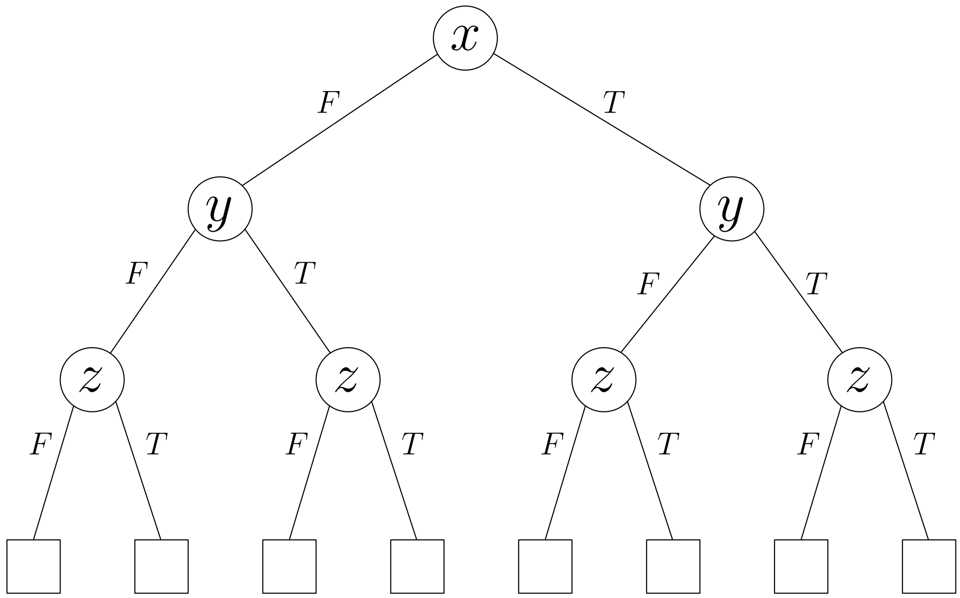



| $x$ | $z$ | $x \vee z$ |
|---|---|---|
| $F$ | $F$ | |
| $F$ | $T$ | |
| $T$ | $F$ | |
| $T$ | $T$ | |

b. Draw the reduced ordered binary decision diagram (ROBDD) for $f$.

**Part II.** Consider the propositional formula $g \equiv (\neg x \wedge y) \vee (\neg x \wedge z)$.

a. Fill in the terminal nodes for the ordered binary decision diagram for $g$ using the variable ordering $x > y > z$. (You may find it helpful to write out the truth table.)




| $x$ | $y$ | $z$ | $(\neg x \wedge y) \vee (\neg x \wedge z)$ |
|---|---|---|---|
| $F$ | $F$ | $F$ | |
| $F$ | $F$ | $T$ | |
| $F$ | $T$ | $F$ | |
| $F$ | $T$ | $T$ | |
| $T$ | $F$ | $F$ | |
| $T$ | $F$ | $T$ | |
| $T$ | $T$ | $F$ | |
| $T$ | $T$ | $T$ | |

b. Draw the reduced ordered binary decision diagram (ROBDD) for $g$.

**Part III.** Construct the reduced ordered binary decision diagram (ROBDD) for the conjunction of $f$ and $g$, that is $f \wedge g$, by combining the ROBDDs for $f$ and $g$. Assume the variable ordering $x > y > z$. Show as many intermediate steps as necessary. (While you may find it helpful to compute the truth table for checking your answer, given here for your convenience, you should arrive at your answer by combining the two ROBDDs.)

| $x$ | $y$ | $z$ | $f \wedge g \equiv (x \vee z) \wedge ((\neg x \wedge y) \vee (\neg x \wedge z))$ |
|---|---|---|---|
| $F$ | $F$ | $F$ | |
| $F$ | $F$ | $T$ | |
| $F$ | $T$ | $F$ | |
| $F$ | $T$ | $T$ | |
| $T$ | $F$ | $F$ | |
| $T$ | $F$ | $T$ | |
| $T$ | $T$ | $F$ | |
| $T$ | $T$ | $T$ | |

**Function** : DPLL($\phi$)
**Input** : CNF formula $\phi$ over $n$ variables
**Output** : true or false, the satisfiability of F
**begin**
    UnitPropagate($\phi$)
    **if** $\phi$ has false clause **then return** false
    **if** all clauses of $\phi$ satisfied **then return** true
    x $\leftarrow$ SelectBranchVariable($\phi$)
    **return** DPLL($\phi[x \mapsto \textit{true}]$) $\vee$ DPLL($\phi[x \mapsto \textit{false}]$)
**end**

Figure 1: The DPLL satisfiability checking algorithm.

**Function** : DPLL($\phi, t$)
**Input** : CNF formula $\phi$ over $n$ variables; $t \in \mathbb{Z}$
**Output** : $\#\phi$, the model count of $\phi$
**begin**
    UnitPropagate($\phi$)
    **if** $\phi$ has false clause **then return** 0
    **if** all clauses of $\phi$ satisfied **then return** $2^t$
    x $\leftarrow$ SelectBranchVariable($\phi$)
    **return** DPLL($\phi[x \mapsto \textit{true}], t-1$) $+$ DPLL($\phi[x \mapsto \textit{true}], t-1$)
**end**

Figure 2: The DPLL satisfiability checking algorithm can be easily converted into a model-counting algorithm.

**Problem 2. Implementing DPLL.** In this problem, I am asking you to implement the DPLL algorithm for satisfiability checking and model counting. Recall that the satisfiability checking algorithm can be easily converted into a model counting algorithm.

In this problem, you will need to implement several functions to build up to the DPLL algorithm. There are a few provided files and some useful functions that you can use.

**Provided files and functionality.**

i. `dpll.py`. This is the main file where you will implement the DPLL algorithm.

ii. `propositional_logic.py`. This is the solution propositional logic file from HW-01. It contains a bunch of useful functions that are demonstrated in `using_propositional_logic.py`.

iii. `random_expression.py`. This is a utility for generating random Boolean expressions. Its usage is demonstrated in `using_propositional_logic.py`

iv. `using_propositional_logic.py`. This file imports the other files and uses the relevant functions. You should take a look at this file an understand what is going on in here. As you are developing, this file is meant to be a test bed for trying things out and seeing what happens.

v. `test_propositional_logic.py`. This is the updated test file from HW-01. They should all pass right now.

vi. `test_dpll.py`. These are tests for the DPLL functions and helper functions. At first, they will not pass because nothing is implemented.

vii. `if_then_else_programs.py`. This is for the next part of the assignment. Don't worry about it yet.

viii. `program_equivalence.py`. This is for the next part of the assignment. Don't worry about it yet.

**Some Background.**

The DPLL algorithm operates on a formula that is in conjunctive normal form (CNF), meaning that it is a conjunction of disjuncts of literals. For instance, this formula is in CNF.

$$(x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$$

However, the DPLL algorithm operates on a list of clauses, where each list is a list of disjuncts, and each disjunct is a literal. Thus, we can rewrite the above formula, in our now familiar Python-based Boolean logic language as a a list of 5 clauses:

```
[[BoolVar(x), BoolVar(y)],
[Not(BoolVar(x)), BoolVar(z)],
[BoolVar(z), BoolVar(w)],
[BoolVar(x)],
[BoolVar(y), BoolVar(v)]]
```

For this assignment, I have give you a function which converts a formula into CNF list form. It is demonstrated in the `using_propositional_logic.py` file, which you can call using `f.cnfListForm()`. I suggest you check out how it works.

Inside of `dpll.py` there are several function for you to implement.

A. `replaceInAllDisjuncts(disjuncts, v, e)`. Recall that in the DPLL algorithm, we need to recursively replace variables with $F$ and $T$.

   I have provided a function called `replace1` that will replace a single variable with an expression. For example, you can write `Not(x).replace1(x, T)` to replace `x` with `T`.

   We are going to start by replacing all occurrences of a variable `v` by expression `e` in a single disjunct, which is a list of literals, and simplifying each.

   For example, replacing `z` with `F`:

```
disjuncts = [BoolVar(z), Not(BoolVar(z)), Not(BoolVar(x))]
replaceInAllDisjuncts(disjuncts, z, F)
[BoolConst(False), BoolConst(True), Not(BoolVar(x))]
```

   Write this function in `dpll.py`, and test it using

   `pytest-3 -k test_replaceInAllDisjuncts`

B. `replaceInAllClauses(clauses, v, e)`. Now we want to replace all occurrences of a variable in each clause in a list of clauses. You should use the function `replaceInAllDisjuncts` that you wrote in the previous step.

   For example, for the example formula given earlier, replacing `x` with `T`:

```
replaceInAllClauses(f.cnfListForm(), x, T)

[[BoolConst(False), BoolConst(True)],
 [BoolConst(False), Not(BoolVar(z))],
 [BoolVar(z), BoolConst(True)],
 [BoolVar(z), Not(BoolVar(z)), BoolConst(False)],
 [BoolConst(True)],
 [BoolConst(True), Not(BoolVar(z)), BoolConst(False)]]
```

   Write this function in `dpll.py`, and test it using

   `pytest-3 -k test_replaceInAllClauses`

C. `allDisjunctsUNSAT(disjuncts)`. Eventually, we want to test if there is a false clause among our list of clauses. The only way a clause can be false is if all of its disjuncts are false (or simplify to false). This function will test if this is the case.

For example:

```
allDisjunctsUNSAT([T,F,F])
False

allDisjunctsUNSAT([F,Not(T),F])
True
```

Write this function in `dpll.py`, and test it using

`pytest-3 -k test_allDisjunctsUNSAT`

D. `containsUNSATClause(clauses)`. Now, we can actually implement the step of the DPLL algorithm that checks if there is an UNSAT clause among the list of clauses. You should use the function you wrote in the previous step to test if there is an UNSAT clause among the list of clauses.

For example:

```
clauses =
[[Not(BoolVar(x)), BoolVar(x)],
 [Not(BoolVar(x)), Not(BoolVar(z))],
 [BoolVar(z), BoolVar(x)],
 [BoolVar(z), Not(BoolVar(z)), Not(BoolVar(x))],
 [BoolVar(x)],
 [BoolVar(x), Not(BoolVar(z)), Not(BoolVar(x))]]

containsUNSATClause(replaceInAllClauses(clauses, x, F))
True

containsUNSATClause(replaceInAllClauses(clauses, z, F))
False
```

Write this function in `dpll.py`, and test it using

`pytest-3 -k test_containsUNSATClause`

E. `someDisjunctSatisfied(disjuncts)`. Eventually, we want to check if all clauses in a list are satisfied. Let's break this into smaller steps to check if a single clause (which is a list of disjuncts) contains a satisfied clause. For example,

```
clause = [Not(x), y, Not(F), z]
someDisjunctSatisfied(clause)
True

clause = [Not(x), F, Not(F), z]
someDisjunctSatisfied(clause)
True

clause = [Not(x), y, Not(w), z]
someDisjunctSatisfied(clause)
False
```

Write this function in `dpll.py`, and test it using

```
pytest -k test_someDisjunctSatisfied
```

F. `allClausesSatisfied(clauses)`. You should now be able to check

```
clauses =
[[BoolVar(x), BoolVar(y)],
 [Not(BoolVar(x)), BoolVar(z)],
 [BoolVar(z), BoolVar(w)],
 [BoolVar(x)],
 [BoolVar(y), BoolVar(v)]]

allClausesSatisfied(clauses)
False

clauses =
[[BoolConst(True), BoolVar(y)],
 [BoolConst(False), BoolConst(True)],
 [BoolConst(True), BoolVar(w)],
 [BoolConst(True)],
 [BoolVar(y), BoolConst(True)]]

allClausesSatisfied(clauses)
True
```

Write this function in `dpll.py`, and test it using

```
pytest -k test_allClausesSatisfied
```

G. `dpll_sat(f)`. This function is actually implemented for you. All it does is get the list of clauses and the variables, then pass them to a helper function to do the actual DPLL work. You'll implement the helper function in the next step.

```
def dpll_sat(f):
    varlist = f.getVars()
    clauses = f.cnfListForm()
    return dpll(clauses, varlist)
```

H. `dpll(clauses, varlist)`. This is where all the action happens. Assuming you have correctly implemented the previous parts, you should be able to implement the DPLL algorithm as given in Figure 1.

**NOTE:** I am not asking you to do the unit propagation step. DPLL should work fine without it. It is an optimization. If you want to give it a try, go for it!

In the step in which a branch variable is selected, the recommended choice is to use the first variable, then pass the remaining variables to the recursive call. Note that the DPLL algorithm as stated does not pass a variable list down the recursive call. Our implementation does indeed pass the variable list along with the list of clauses. You just trust me that this is easier, or you can think about what you would otherwise do.

Implement this function in `dpll.py`. There are three ways to test it:

- `pytest -k test_dpll_sat` will run a basic quite of tests.
- `pytest -k test_dpll_sat_vs_tt_sa` will run your DPLL implementation and compare its result to a brute force SAT check by generating the truth table and searching for a row that evaluates to True.
- You can randomly generate expressions and compare the results of the truth table method and DPLL for the random expressions.

```
from random_expression import *

#do 100 random tests
random_sat_test(100)
```

I. `equiv_dpll(f1, f2)`. You should now be able to check if two formulas are logically equivalent using the DPLL method you implemented. Implement this function in `dpll.py`. You might want write your own tests for this.

**Problem 3. Model Counting.** Now that you have implemented the DPLL satisfiability checking algorithm, you should be able to convert it directly into a model counting algorithm (See Figure 2).

A. `dpll_model_count(f)`. This function is given. It is similar to the main DPLL satisfiability checking algorithm. It will call the next helper function to do all the real work.

```
def dpll_model_count(f):
varlist = f.getVars()
clauses = f.cnfListForm()
return dpll_count(clauses, varlist, len(varlist))
```

B. `dpll_count(clauses, varlist, t)`. Implement this counting function as in Figure 2. When you are done, you can test this in two ways:

- `pytest -k test_dpll_model_count_vs_tt_count` will compare your DPLL counting algorithm to a brute-force truth table method.
- You can randomly generate expressions and compare the counts from the truth table method and DPLL for the random expressions.

```
from random_expression import *

#do 100 random counting tests
random_count_test(100)
```

**Problem 4. Checking Program Equivalence.** We can check for equivalence of programs in a very simple programming language using Boolean satisfiability checking. Imagine an imperative programming language that has and If-Else conditional control flow structure and allows functions calls. We will not worry about what the functions do; we are abstracting that away and just looking at the form of the program and the names of the variables and functions.

Our goal is to convert programs into Boolean expressions and then check if the resulting Boolean expressions are equivalent. Shown here are three simple programs.

**Program 1**

```
If((!A & !B))
  Call h
else
  If(!A)
    Call g
  else
    Call f
```

**Program 2**

```
If(A)
  Call f
else
  If(!B)
    Call g
  else
    Call h
```

**Program 3**

```
If(A)
  Call f
else
  If(B)
    Call g
  else
    Call h
```

Converting a program like these into a Boolean expression is straightforward. For a construct of the form `If condition S1 else S2`, where `S1` and `S2` are more program statements, we can encode the program control flow as

$$(\text{condition} \wedge S_1) \vee (\neg\text{condition} \wedge S_2)$$

In the file `program_equivalence.py`, there are programs built from two object constructors

`IfThenElse(condition, then_branch, else_branch)`

`FunctionCall(fun_name)`
where `then_branch` and `else_branch` are Boolean logic expressions as we have seen before. Their definitions are in the file `if_then_else_programs.py`. In this problem, you should

A. implement the `toBool()` functions of the `IfThenElse` and `FunctionCall` classes inside the `if_then_else_programs.py` file. For example, you should get something like this:

```
print(program1.toBool().format())
(((~A & ~B) & h) | (~(~A & ~B) & ((~A & g) | (~~A & f))))
```

B. implement the function `equiv_programs(P1, P2)` inside the `program_equivalence.py` file. You should use your DPLL algorithm, or formula equivalence checking function to implement this function.

Test your code by running the `program_equivalence.py` file and confirming that it does what you think it should do.