# CISC 360 Assignment 3
# due Monday, 2022–03–14 at 11:59pm, via onQ

Jana Dunfield

March 5, 2022

**Reminder:** All work submitted must be your own, or, if you are working with one other student, your teammate's.

**Late policy:** Assignments submitted up to 24 hours late (that is, by 11:59 pm the following day) will be accepted **without penalty**. Assignments submitted more than 24 hours late will **not** be accepted, except with an accommodation or a consideration granted according to policy.

## IMPORTANT: Your file must compile

Your file **must** load (`:load` in GHCi) successfully, or we will subtract **30%** from your mark.

If you are halfway through a problem and run out of time, **comment out the code that is causing :load to fail** by surrounding it with {- ... -}, and write a comment describing what you were trying to do. We can often give (partial) marks for evidence of progress towards a solution, but **we need the file to load and compile**.

## If you choose to work in a group of 2

You **must** use version control (such as GitHub, GitLab, Bitbucket, etc.). This is primarily to help you maintain an equitable distribution of work, because commit logs provide information about the members' level of contribution.

Your repository **must** be private—otherwise, anyone who has your GitHub (etc.) username can copy your code, which would violate academic integrity. However, upon request from the course staff, you must give us access to your repository. (You do not need to give us access unless we ask.)

We only need *one* submission of the assignment ("Assignment 3"). However, each of you *must* submit a brief statement.

1. Give your names and student ID numbers.
2. Estimate the number of hours you spent on the assignment.
3. Briefly describe your contribution, and your teammate's contribution. (Coding, trying to understand the assignment, testing, etc.)

This is meant to ensure that both group members reflect on their relative contributions.

If you do not submit a statement, you will not receive an assignment mark. This is meant to ensure that each group member is at least involved enough to submit a statement. **Each** member must submit a statement.

# 1 Add your student ID

The file `a3.hs` will not compile until you add your student ID number by writing it after the `=`:

```
-- Your student ID:

student_id :: Integer
student_id =
```

You do not need to write your name. When we download your submission, onQ includes your name in the filename.

If you are working in a group of 2, put the second student ID in a comment, like this:

```
-- Your student ID:
student_id :: Integer
student_id = 11112222    -- 33334444
```

# 2 Q2: Truth tables

## 2.1 Formulas

In this assignment, we represent a logical formula as the data type `Formula`. Every `Formula` has one of the following forms:

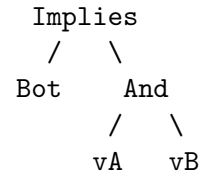
- `Top`, representing truth (sometimes written $\top$)
- `Bot`, representing falsehood (sometimes written $\bot$)
- `And phi1 phi2`, representing conjunction ($\varphi_1 \wedge \varphi_2$)
- `Implies phi psi`, representing implication ($\varphi \to \psi$ or $\varphi \supset \psi$: "if $\varphi$ (phi) then $\psi$ (psi)")
- `Atom v`, representing a propositional variable ("atomic formula", or "propositional constant") named `v`.

```
type Variable = String

data Formula = Top                      -- truth (always true)
             | Bot                      -- falsehood (contradiction) (not used in Q3)
             | And Formula Formula      -- conjunction
             | Or Formula Formula       -- disjunction
             | Implies Formula Formula  -- implication
             | Not Formula              -- negation (not used in Q3)
             | Atom Variable            -- atomic proposition ("propositional variable")
             deriving (Eq, Show)
```

For example, if `vA`, `vB` and `formula2` are defined as follows:

```
vA = Atom "A"
vB = Atom "B"

formula2 = Implies Bot (And vA vB)
```

then `formula2` represents "if false then (A and B)", and can be drawn as the tree




## 2.2 Valuations and truth tables

A `Valuation` maps each atomic proposition variable to a Boolean:

```
type Valuation = [(Variable, Bool)]
```

For example, the valuation `[("A", False), ("B", True)]` indicates that `Atom "A"` is considered false, and `Atom "B"` is considered true. Under this valuation, the formula `And vA vB` would be false (because `vA` is `False` in the valuation), but the formula `Implies vA vB` would be true (because `vB` is true in the valuation).

## 2.3 Q2a: getAtoms

Complete the function `getAtoms`, which gathers all the atomic propositions (like `"A"`, `"B"`, etc.) that appear in a formula. `getAtoms` should never return any duplicates: given the formula `And vA vA`, that is, `And (Atom "A") (Atom "A")`, your implementation of `getAtoms` should return `["A"]`, not `["A", "A"]`.

(Hint: Read the comment in `a3.hs` about the Haskell library function `nub`.)

```
getAtoms :: Formula -> [Variable]
```

## 2.4 Q2b: getValuations

Complete the function `getValuations`, which—given a list of variables (strings)—returns *all* possible valuations of `True` and `False`.

For example, `getValuations ["A", "B"]` should return a list containing the four possible valuations:

1. one valuation in which both `"A"` and `"B"` are `True`
2. one valuation in which both `"A"` and `"B"` are `False`
3. one valuation in which `"A"` is `True` and `"B"` is `False`
4. one valuation in which `"A"` is `False` and `"B"` is `True`

The order of the valuations in the output doesn't matter.

```
getValuations :: [Variable] -> [Valuation]
```

**Hint:** Think carefully about the base case when the argument (list of variables) is empty. If there are no variables, there is one possible valuation, which is the empty valuation (empty list). That is *not* the same as saying there are no valuations; if there are no valuations, you should return the empty list, but if there is one possible valuation you should return a list *whose only element* is that one valuation.

### 2.5 Q2c: evalF

Complete the function `evalF` which, given a valuation `valu` and a formula `phi`, evaluates the formula `phi` as follows:

1. If `phi` is `Top`, then `evalF` returns `True`. (Already written for you.)
2. If `phi` is `Bot`, then `evalF` returns `False`. (Already written for you.)
3. If `phi` is `Not psi`, then `evalF` returns the negation of `evalF psi`. (Already written for you.)
4. If `phi` is `Atom v`, then `evalF` should look up `v` in the valuation `valu`, and return the associated boolean. For example, `evalF [("B", False)] (Atom "B")` should return `False`.
5. If `phi` is `And phi1 phi2`, then `evalF` should return `True` if and only if `evalF` returns `True` for both `phi1` and `phi2` (under the same valuation `valu`).
6. If `phi` is `Or phi1 phi2`, then `evalF` should return `True` if and only if `evalF` returns `True` for either `phi1` or `phi2` (under the same valuation `valu`).
7. If `phi` is `Implies phi1 phi2`, then `evalF` should return `True` if and only if *either*
   - `evalF` returns `False` on `phi1`, *or*
   - `evalF` returns `True` on `phi2`.

   (This is the same idea as the CISC 204 "material implication" equivalence $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$.)

   As with `And` and `Or`, use the same valuation `valu` in both recursive calls.

### 2.6 Testing Q2

Once you have completed all three functions above, you can use our function `buildTable` to test them by printing a truth table with all possible valuations. For example, calling `buildTable` on `And vA vB` should produce a table with four rows, with all possible valuations of A and B, such that the right-hand "result" is `True` only when both A and B are true.

## 3 Q3: Tiny Theorem Prover

This question uses some of the same types as Q2, but in a rather different way.

You will implement a function `prove` that takes a *context* containing formulas that are *assumed to be true*. If the formula can be proved *according to the rules given below* then `prove` should return `True`. Otherwise, it should return `False`.

If `phi` is a formula, and `ctx` is a context (list of assumptions), then we write

$$\texttt{ctx} \vdash \texttt{phi}$$

to mean that `phi` is provable ("true") under the assumptions in `ctx`

A warning: the notation in this question varies substantially from that in CISC 204. The notation used in this problem is similar to that used in the second half of Gentzen's dissertation—which is *not* the half that CISC 204's "natural deduction" comes from. The second half of Gentzen's dissertation describes *sequent calculus*. Like 204, sequent calculus includes sequents; a sequent is a list of premises (written before the "turnstile" symbol $\vdash$) along with a conclusion (written after the $\vdash$). Unlike 204, the rules of sequent calculus operate by manipulating the sequent, rather than by constructing a line-numbered proof.

I have intentionally omitted rules for falsehood (`Bot` / $\bot$) and negation: some of those rules require guessing and forward reasoning, making them quite difficult to implement.

First, let's look at some examples. (These are just examples. To correctly implement this question, you need to understand and follow the inference rules in Figure 1.)

1. `[]` $\vdash$ `Top` is provable, because `Top` is *always* true. (In fact, `ctx` $\vdash$ `Top` is provable regardless of what assumptions are in `ctx`.)

2. `[Atom "A"]` $\vdash$ `Atom "A"` is provable, because we are assuming the propositional variable A is true.

3. `[Atom "A"]` $\vdash$ `Atom "B"` is *not* provable, because knowing that the variable A is true tells us nothing about whether B is true.

4. `[Atom "A", Atom "B"]` $\vdash$ `And (Atom "B") (Atom "A")` is provable, because we are assuming A, and assuming B.

5. `[And (Atom "A") (Atom "B")]` $\vdash$ `And (Atom "B") (Atom "A")` is provable, because we are assuming $A \wedge B$, and the only way that $A \wedge B$ could be true is if both $A$ and $B$ are true. Therefore, $B$ is true, and $A$ is true, so $B \wedge A$ is true.

6. `[]` $\vdash$ `Implies (Atom "A") (Atom "A")` is provable:

   (a) To see if `[]` $\vdash$ `Implies (Atom "A") (Atom "A")` is true, we suppose (assume) the antecedent of the implication, which is `Atom "A"`, and then prove the consequent, `Atom "A"`. We do this by "reducing" the problem to

   $$\texttt{[Atom "A"]} \vdash \texttt{[Atom "A"]}$$

   (A difference between sequent calculus and 204's natural deduction is that, while premises and assumptions are pretty similar in natural deduction, they aren't considered the same. For example, you might lose marks in 204 if you claimed you were using a premise when you were actually using an assumption introduced by the rule $\rightarrow$i. Sequent calculus does not distinguish these: an assumption added using "Implies-Right", which is the sequent calculus equivalent to $\rightarrow$i, is added to the list of premises in the sequent.)

7. `[Implies (Atom "A") (Atom "B"),Atom "A"] ⊢ Atom "B"` is provable: we have two assumptions, one that A implies B, and a second assumption that A is true. By these assumptions, B is true. To show this, we reduce the problem to

   $$\texttt{[Atom "B",Atom "A"]} \vdash \texttt{Atom "B"}$$

   Why can we do that? We are assuming that A implies B, but we know A (because we are assuming it). So we produce some new knowledge—that B is true—and add it to the assumptions.

   *In this specific setting*, it is legitimate to *replace* the assumption `Implies (Atom "A") (Atom "B")` with `Atom "B"`, which is the consequent of the implication.

Expressed as *inference rules*, the rules we want to implement are shown in Figure 1.

$$\frac{\texttt{phi} \in \texttt{ctx}}{\texttt{ctx} \vdash \texttt{phi}}\ \text{UseAssumption} \qquad \frac{}{\texttt{ctx} \vdash \texttt{Top}}\ \text{Top-Right}$$

$$\frac{\texttt{ctx} \vdash \texttt{phi1} \qquad \texttt{ctx} \vdash \texttt{phi2}}{\texttt{ctx} \vdash \texttt{And phi1 phi2}}\ \text{And-Right} \qquad \frac{\texttt{phi} : \texttt{ctx} \vdash \texttt{psi}}{\texttt{ctx} \vdash \texttt{Implies phi psi}}\ \text{Implies-Right}$$

$$\frac{\texttt{ctx} \vdash \texttt{phi1}}{\texttt{ctx} \vdash \texttt{Or phi1 phi2}}\ \text{Or-Right-1} \qquad \frac{\texttt{ctx} \vdash \texttt{phi2}}{\texttt{ctx} \vdash \texttt{Or phi1 phi2}}\ \text{Or-Right-2}$$

$$\frac{\texttt{ctx1 ++ [phi1] ++ ctx2} \vdash \texttt{phi} \qquad \texttt{ctx1 ++ [phi2] ++ ctx2} \vdash \texttt{phi}}{\texttt{ctx1 ++ [Or phi1 phi2] ++ ctx2} \vdash \texttt{phi}}\ \text{Or-Left}$$

$$\frac{\texttt{ctx1 ++ [phi1, phi2] ++ ctx2} \vdash \texttt{phi}}{\texttt{ctx1 ++ [And phi1 phi2] ++ ctx2} \vdash \texttt{phi}}\ \text{And-Left}$$

$$\frac{\texttt{ctx1 ++ ctx2} \vdash \texttt{phi1} \qquad \texttt{ctx1 ++ [phi2] ++ ctx2} \vdash \texttt{psi}}{\texttt{ctx1 ++ [Implies phi1 phi2] ++ ctx2} \vdash \texttt{psi}}\ \text{Implies-Left}$$

**Figure 1 Rules to implement within `prove'` and `prove_left`**

We attempt to explain these rules in English. In each rule, the conclusion of the rule is below the line, and the premises (if any) are above the line.

1. 'UseAssumption' says that if we are trying to prove a formula, and the formula appears in ($\in$) our list of assumptions, then it is provable: we have assumed exactly that formula.
2. 'Top-Right' says that the true formula is always provable.
3. 'And-Right' says that a conjunction is provable if both *subformulas* `phi1` and `phi2` are provable (And-Right has one premise for `phi1`, and one premise for `phi2`).

   This corresponds to the $\wedge$i rule in CISC 204.

4. 'Implies-Right' says that an implication is provable if, assuming the antecedent, we can prove the consequent. The premise is written using Haskell's "cons" notation: we add the antecedent `phi` to the list of assumptions `ctx` and try to prove the consequent `psi`.

   This corresponds to the →i rule in CISC 204.

5. 'Or-Left' says that if we're assuming a disjunction `Or phi1 phi2` is true, we can "case-split" on it: prove the goal under assumption `phi1`, and also under assumption `phi2`.

   This corresponds to the ∨e rule in CISC 204.

6. 'Or-Right-1' says that a disjunction is provable if its first subformula, `psi1`, is provable.

   This corresponds to the ∨i1 rule in CISC 204.

7. 'Or-Right-2' says that a disjunction is provable if its second subformula, `psi2`, is provable.

   This corresponds to the ∨i2 rule in CISC 204.

8. 'And-Left' says that if we are *assuming* a conjunction, we should "decompose" the conjunction to bring out each of the subformulas `phi1` and `phi2` as a separate assumption.

   This corresponds roughly to the ∧e1 *and* ∧e2 rules being used simultaneously in CISC 204 (if that were allowed).

9. 'Implies-Left' says that if we are *assuming* that `phi1` implies `phi2`, *and* (first premise) the antecedent `phi1` is provable, *and* (second premise) we can prove `psi` assuming `phi2`, then `psi` is provable under our original assumptions `ctx1 ++ [Implies phi1 phi2] ++ ctx2`.

   This corresponds roughly to the →e rule in CISC 204.

A general explanation of how to turn inference rules into code is beyond the scope of this assignment, and this particular set of rules is subtle, so we have given you a framework to build upon.

This framework uses *continuations* to handle the "search" aspect of the problem: we are *searching* for a proof using the rules in Figure 1.

**Note:** As usual, we may test your code using additional test cases that we do not make available to you.

**CISC 204 notes:** The function `prove'` implements *backward reasoning*: it looks at the shape of the conclusion (the formula after the turnstile symbol "⊢"), and tries to do the last step of the proof, which tells it what the next-to-last step in the conclusion should be.

The function `prove_left` implements *forward reasoning* on premises (assumptions).

The job of alternating between backward and forward reasoning is implemented in the structure of the two functions `prove'` and `prove_right`.