CISC 360 Assignment 5
due Wednesday, 2021–12–01 at 11:59pm, via onQ

Jana Dunfield

November 22, 2021

This assignment is essentially a3, but in Prolog instead of Haskell.

Reminder: All work submitted must be your own, or, if you are working with one other student, yours and 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.

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. (But you do not need to give us access unless we ask.)

We only need *one* submission of the assignment ("Assignment 4"). However, each of you *must* submit a brief statement ("Assignment 4 Group Statements"):

1. Estimate the number of hours you spent on the assignment.
2. 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. That is, you must make two separate submissions in "Assignment 4 Group Statements".

Overview

This assignment is essentially a Prolog version of a3.

The last part of a3 implemented what CISC 204 calls "backwards reasoning" to prove the truth of formulas under assumptions. Prolog is a good match for this, because Prolog has backwards reasoning built-in. Some of the machinery in a3, like the `decompose` function, becomes much simpler in Prolog.

IMPORTANT: Do not submit a .rar file

Because onQ does not like filenames ending in `.pl`, you may put your a4.pl file into a `.zip` archive.

Do not submit other archive formats. In particular, do *not* submit a `.rar` file. Not all of the course staff have the software to unpack that format.

If we are not able to unpack your archive, your assignment will not be marked.

Therefore, **submit a `.zip` file.** Do not submit a `.rar` file.

IMPORTANT: Your file must compile

Your file **must** load (`consult` in SWI-Prolog) successfully, or we will subtract **30%** from your mark.

If you are halfway through a problem and run out of time, **comment out the problematic code** by surrounding it with `/*...*/` and add a comment describing what you were trying to do. We generally give (partial) marks for evidence of progress in solving a problem, but **we need the file to load**.

It is your responsibility to submit the right version of the file.

(Warnings about singleton variables do **not** prevent the file from loading.)

Late policy: Assignments submitted up to 24 hours late (that is, by 11:59 pm the following day) will be accepted **with a 15% penalty**.

1 Add your student ID

Begin by adding your student ID number in `a5.pl`, after `student_id(`, replacing "`this is a syntax error`".

```
/*
 * Q1: Student ID
 */
student_id( this is a syntax error ).
% other_student_id(  ).
% if in a group, uncomment the above line and put the other student ID here
```

2 Q2: Truth tables, in Prolog

2.1 Formulas

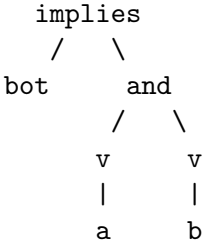
In this assignment, we represent a logical formula using Prolog atoms. Every formula is one of the following:

- `top`, representing truth (written $\top$ in 204)
- `bot`, representing falsehood (written $\bot$ in 204)
- `and(P1,P2)`, representing conjunction P1 $\wedge$ P2
- `or(P1,P2)`, representing disjunction P1 $\vee$ P2
- `implies(P,Q)`, representing implication P $\rightarrow$ Q: "if P then Q"
- `not(P)`, representing negation: $\neg$P
- `equiv(P, Q)`, representing equivalence: P $\Leftrightarrow$ Q: "P if and only if Q"
- `v(V)`, representing an atomic proposition `V`, where `V` is an atom like `a`, `b` or `d`

For example,

```
implies(bot, and(v(a), v(b)))
```

represents "if false then (a and b)", and can be drawn as the tree



2.2 Valuations and Truth Tables

A valuation is a list of pairs `(C, B)` that maps each atomic formula (variable) to a Boolean.

For example, the valuation `[(a, false), (b, true)]` indicates that `v(a)` is considered false, and `v(b)` is considered true. Under this valuation, the formula `and(v(a), v(b))` would be false (because `a` is `false` in the list), but `implies(v(a), v(b))` would be true (because `b` is true in the list).

2.3 Q2a: getAtoms

Complete the predicate `getAtoms`, which "gathers" all the variables (`a`, `b`, etc.) that appear in a formula. `getAtoms` should not return any duplicates: given the formula `and(v(a), v(a))`, your implementation of `getAtoms` should return `[a]`, not `[a, a]`.

```
?- getAtoms(Formula, Vars).
```

For example, the following queries should be `true`:

```
?- getAtoms(and(v(a), v(a)), [a]).
?- getAtoms(and(v(b), v(a)), [b, a]).
?- getAtoms(and(implies(v(a), v(d)), implies(bot, v(b))), [a, d, b]).
```

2.4 Q2b: oneValuation

Complete the predicate `oneValuation`, which—given a list of propositional variables—"returns" one possible valuation to that list of variables. You can ask Prolog to find all the possible valuations by typing a semicolon (";").

For example (typing ; after each line):

```
?- oneValuation([a, b], Valuation).
Valuation = [(a, true), (b, true)]
Valuation = [(a, true), (b, false)]
Valuation = [(a, false), (b, true)]
Valuation = [(a, false), (b, false)].
```

Hint: Use `is_bool`.

2.5 getValuations

There's nothing for you to do here, but we wanted to mention that we translated Haskell `getValuations` to Prolog, if you want to see how that works.

The predicate `getValuations`, given a list of propositional variables, "returns" *all* possible valuations of `true` and `false`.

For example:

```
?- getValuations([a, b], Valuations).
Valuations = [[(a, true), (b, true)],
              [(a, true), (b, false)],
              [(a, false), (b, true)],
              [(a, false), (b, false)]]
```

2.6 Q2c: evalF

Complete the predicate `evalF` which, given a formula `Q` and a valuation `Valu`, "returns" either `true` or `false`:

1. If `Q` is `top`, then `evalF` returns `true`. (Already written for you.)
2. If `Q` is `bot`, then `evalF` returns `false`. (Already written for you.)
3. If `Q` is `v(V)`, then `evalF` should look up `V` in the valuation `Valu`, and return the associated boolean. (Already written for you). For example:

```
?- evalF(v(b), [(b, false)], false).
true.
```

4. If `Q` is `and(Q1, Q2)`, then `evalF` should return `true` if `evalF` returns `true` for both `Q1` and `Q2` (under the same valuation `Valu`).

   (Already written for you).

5. If `q` is `implies(Q1, Q2)`, then `evalF` should return `true` if *either*

   - `evalF(Q1, Valu, false)`, *or*
   - `evalF(Q2, Valu, true)`.

6. If `q` is `or(Q1, Q2)`, then `evalF` should return `true` if *either*

   - `evalF(Q1, Valu, true)`, *or*
   - `evalF(Q2, Valu, true)`.

   As with `and`, use the same valuation `Valu` in the recursive calls.

7. If `Q` is `not(Q1)`, then `evalF` should return `true` if `evalF(Q1, Valu, false)`, and `false` otherwise.

8. If `Q` is `equiv(Q1, Q2)`, then `evalF` should return `true` if `Q1` and `Q2` evaluate to the same formula (under the same valuation `Valu`).

2.7 Testing Q2

Once you have completed `evalF`, you can use our predicate `buildTable` to test them by printing a truth table with all possible valuations. For example:

```
?- buildTable(and(v(a), v(b)), _).
formula: and(v(a),v(b))
'a = true, b = true, ':   true
'a = true, b = false, ':   false
'a = false, b = true, ':   false
'a = false, b = false, ':   false
true
```

3 Q3: Tiny Theorem Prover

You will implement a predicate `prove` that takes a *context* `Ctx` containing formulas that are *assumed to be true*. If the formula `P` can be proved *according to the rules given below* then `prove` should be satisfied, that is, the query `prove(Ctx, P)` should be `true`.

If P is a formula, and Ctx is a context (list of assumptions), then we write

$$\text{Ctx} \vdash \text{P}$$

to mean that P is provable ("true") under the assumptions in `Ctx`. For example:

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

2. [] ⊢ `bot` is not provable, because `bot` is false and we have no assumptions.

3. [`bot`] ⊢ `v(a)` is provable: if we *assume* that `bot` is true, then anything (such as `v(a)`) is true.

4. [`v(a)`] ⊢ `v(a)` is provable, because we are assuming the propositional variable `a` is true.

5. [`v(a)`] ⊢ `v(b)` is *not* provable, because knowing that `a` is true tells us nothing about whether `b` is true.

6. [`v(a)`, `v(b)`] ⊢ `and(v(b), v(a))` is provable, because we are assuming `a`, and assuming `b`.

7. [`and(v(a), v(b))`] ⊢ `and(v(b), v(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.

8. [] ⊢ `implies(v(a), v(a))` is provable:

   (a) To see if [] ⊢ `implies(v(a), v(a)` is true, we suppose (assume) the antecedent of the implication, which is `v(a)`, and then prove the consequent, `v(a)`. We do this by "reducing" the problem to

   [`v(a)`] ⊢ [`v(a)`]

9. [`implies(v(a), v(b))`, `v(a)`] ⊢ `v(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

   [`v(b)`, `v(a)`] ⊢ `v(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(v(a), v(b)` with `v(b)`, which is the consequent of the implication. (This does not hold for many specific logics, but I am confident it holds for this one. If I am wrong, you will still get full marks for implementing this.)

Expressed as *inference rules*, the rules to implement are:

$$\frac{P \in Ctx}{Ctx \vdash P}\ \text{UseAssumption}$$

$$\frac{}{Ctx \vdash \texttt{top}}\ \text{Top-Right} \qquad \frac{Ctx \vdash Q1 \qquad Ctx \vdash Q2}{Ctx \vdash \texttt{and}(Q1, Q2)}\ \text{And-Right} \qquad \frac{[P \mid Ctx] \vdash Q}{Ctx \vdash \texttt{implies}(P, Q)}\ \text{Implies-Right}$$

$$\frac{[Q1 \mid Ctx] \vdash Q2 \qquad [Q2 \mid Ctx] \vdash Q1}{Ctx \vdash \texttt{equiv}(Q1, Q2)}\ \text{Equiv-Right}$$

$$\frac{}{Ctx1 \mathbin{++} [\texttt{bot}] \mathbin{++} Ctx2 \vdash Q}\ \text{Bot-Left} \qquad \frac{Ctx1 \mathbin{++} [P1, P2] \mathbin{++} Ctx2 \vdash Q}{Ctx1 \mathbin{++} [\texttt{and}(P1, P2)] \mathbin{++} Ctx2 \vdash Q}\ \text{And-Left}$$

$$\frac{Ctx1 \mathbin{++} Ctx2 \vdash P1 \qquad Ctx1 \mathbin{++} [P2] \mathbin{++} Ctx2 \vdash Q}{Ctx1 \mathbin{++} [\texttt{implies}(P1, P2)] \mathbin{++} Ctx2 \vdash Q}\ \text{Implies-Left}$$

$$\frac{Ctx1 \mathbin{++} [\texttt{implies}(P1, P2), \texttt{implies}(P2, P1)] \mathbin{++} Ctx2 \vdash Q}{Ctx1 \mathbin{++} [\texttt{equiv}(P1, P2)] \mathbin{++} Ctx2 \vdash Q}\ \text{Equiv-Left}$$

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* Q1 and Q2 are provable (And-Right has one premise for Q1, and one premise for Q2).
4. 'Implies-Right' says that an implication is provable if, assuming the antecedent, we can prove the consequent. The premise is written using Prolog's "cons" notation: we add the antecedent P to the list of assumptions Ctx and try to prove the consequent Q.
5. 'Equiv-Right' says that an equivalence is provable if we can prove the right side of the formula assuming the left side, and prove the left side assuming the right side.
6. 'Bot-Left' says that if we are *assuming* falsehood, any formula Q is true.
7. 'And-Left' says that if we are *assuming* a conjunction, we should "decompose" the conjunction to bring out each of the subformulas P1 and P2 as a separate assumption.
8. 'Implies-Left' says that if we are *assuming* that P1 implies P2, *and* (first premise) the antecedent P1 is provable, *and* (second premise) we can prove Q assuming P2, then Q is provable under our original assumptions Ctx1 ++ [implies(P1, P2)] ++ Ctx2.
9. 'Equiv-Left' says that if we can prove Q assuming that P1 implies P2, and also P2 implies P1, then Q is provable assuming that P1 and P2 are equivalent.

Unlike the Haskell version of this question, *you don't need to follow an algorithm*: instead of (1) calling `decompose`, (2) checking if the goal is in the decomposed context, and (3) calling `prove_right`, each inference rule becomes a Prolog clause. The "decomposition" is done by the "-Left" rules.

Q3a. Write Prolog clauses that correspond to the rules Bot-Left, And-Right and Implies-Right. There is no "starter code" for these, but looking at our Prolog clause for 'UseAssumption' may be helpful.

Q3b. Implement Implies-Left and Equiv-Left.
Hint: Compare our Prolog clause for 'And-Left' to the rule 'And-Left' above.

4 Q4: Add 'Or'

In Haskell, the Or-Left rule is difficult, which is why it was a bonus question on a3. In Prolog, it's not much more difficult than the other -Left rules.

Implement the following inference rules:

$$\frac{Ctx \vdash P1}{Ctx \vdash or(P1, P2)}\ \text{Or-Right1} \qquad \frac{Ctx \vdash P2}{Ctx \vdash or(P1, P2)}\ \text{Or-Right2}$$

$$\frac{Ctx1 \mathbin{++} P1 \mathbin{++} Ctx2 \vdash Q \qquad Ctx1 \mathbin{++} P2 \mathbin{++} Ctx2 \vdash Q}{Ctx1 \mathbin{++} [or(P1, P2)] \mathbin{++} Ctx2 \vdash Q}\ \text{Or-Left}$$

5 Bonus: Add 'Not'

The above questions are collectively worth 100% of the assignment.

This bonus question is worth up to 5% of the main assignment, and may cause your mark to exceed 100%, but this question probably requires much more than 5% of the effort of the main assignment. Please attempt it only if you really want to.

First, develop rules for 'not' (like our rules for Or-Right1, etc., though Or-Right1 is not particularly good inspiration), and write them in a comment. You are free to refer to logic textbooks or other material on logic to design your rules.

Second, explain how you developed the rules. "I took them from my 204 textbook" is an acceptable explanation.

Third, implement them.

Fourth, test them. Give queries (in a comment) that show that your code matches your rules, at least for some test cases.

Adding rules for 'not' may affect how the main questions behave, so you should copy your file, rename it `a5bonus.pl`, and work on 'Not' there.