Problem Set 2

This second problem set explores mathematical logic. We've chosen the questions here to help you get a more nuanced understanding for what first-order logic statements mean (and, importantly, what they don't mean) and to give you a chance to practice your proofwriting. By the time you've completed this problem set, we hope that you have a much better grasp of mathematical logic and how it can help improve your proofwriting structure.

Because of the Dr. Martin Luther King, Jr. holiday, there are a few problems on this problem set that reference concepts we will cover in Monday's lecture. Those questions are clearly marked as such, and anything that doesn't explicitly warn about this can be completed purely using the material up to and including the lecture when this problem set is released. If you'd like to get an early jump on the remaining problems, visit the course website and check out the Guide to Negations and Guide to First-Order Translations, which will provide an overview of the relevant skills.

Before attempting this problem set, we recommend that you do the following:

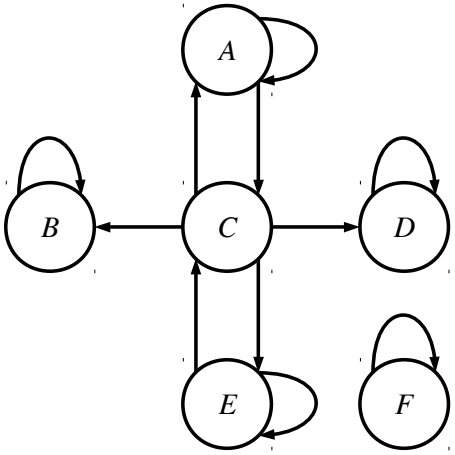
- Familiarize yourself with the online Truth Table Tool and play around with it a bit to get a feel for the propositional connectives.
- Read the online "Guide to Negations" and "Guide to First-Order Translations" (either on Monday, or when this problem set goes out if you want to get a jump on things).
- Read Handout #14, "First-Order Translation Checklist," to get a better sense for common errors in first-order logic translations and how to avoid them. ***We will be running these checklists on your translations, so please be sure to double-check your work before submitting!***

Checkpoint Questions Due Monday, January 22nd at 2:30PM.
Remaining Questions Due Friday, January 26th at 2:30PM.

This week's checkpoint problem comes in the shape of a Google Form, which you can find online using this link. Unlike the other assignments for this quarter, you will not submit this assignment on Grade-Scope, and you'll get feedback about incorrect answers as you go.

Checkpoint Problem: Interpersonal Dynamics (2 Points if Submitted)

The diagram to the right represents a set of people named *A*, *B*, *C*, *D*, *E*, and *F*. If there's an arrow from a person *x* to a person *y*, then person *x* loves person *y*. We'll denote this by writing *Loves*(*x*, *y*). For example, in this picture, we have *Loves*(*C*, *D*) and *Loves*(*E*, *E*), but not *Loves*(*D*, *A*).



There are no "implied" arrows anywhere in this diagram. For example, even though *A* loves *C* and *C* loves *E*, the statement *Loves*(*A*, *E*) is false because there's no direct arrow from *A* to *E*. Similarly, even though *C* loves *D*, the statement *Loves*(*D*, *C*) is false because there's no arrow from *D* to *C*.

At the linked Google Form, you'll find a series of first-order logic formulas about this particular group of people. For each of those formulas, determine whether it's true or false about this group. No justification is necessary.

The remainder of these problems should be completed and submitted through GradeScope by Friday at 2:30PM.

Problem One: Implies and False

Although propositional logic has many different connectives, it turns out that any formula in propositional logic can be rewritten as an equivalent propositional formula that uses only the ¬, ∧, and ⊤ connectives. (You don't need to prove this). In this problem, you will prove a different result: every formula in propositional logic can be rewritten as an equivalent logical formula purely using the → and ⊥ connectives.

i. Find a formula that's logically equivalent to $\neg p$ that uses only the variable p and the → and ⊥ connectives. No justification is necessary.

ii. Find a formula that's logically equivalent to ⊤ that uses only the → and ⊥ connectives. No justification is necessary.

iii. Find a formula that's logically equivalent to $p \wedge q$ that uses only the variables p and q and the → and ⊥ connectives. No justification is necessary.

As a hint, what happens if you negate an implication?

Since you can express ¬, ∧, and ⊤ using just → and ⊥, every possible formula in propositional logic can be expressed using purely the → and ⊥ connectives. Nifty!

Problem Two: Ternary Conditionals

Many programming languages support a *ternary conditional operator*. For example, in C, C++, and Java, the expression x `?` y `:` z means "evaluate the boolean expression x. If it's true, the entire expression evaluates to y. If it's false, the entire expression evaluates to z."

In the context of propositional logic, we can introduce a new ternary connective `?:` such that p `?` q `:` r means "if p is true, the connective evaluates to the truth value of q, and otherwise it evaluates to the truth value of r."

i. Based on this description, write a truth table for the `?:` connective.

ii. Find a propositional formula equivalent to p `?` q `:` r that does not use the `?:` connective. Justify your answer by providing a truth table for your new formula.

It turns out that it's possible to rewrite any formula in propositional logic using only `?:`, ⊤, and ⊥. The rest of this question will ask you to show this.

iii. Find a formula equivalent to $\neg p$ that does not use any connectives besides `?:`, ⊤, and ⊥. No justification is necessary.

iv. Find a formula equivalent to $p \rightarrow q$ that does not use any connectives besides `?:`, ⊤, and ⊥. No justification is necessary.

Since all remaining connectives can be written purely in terms of ¬ and →, any propositional formula using the seven standard connectives can be rewritten using only the three connectives `?:`, ⊤, and ⊥.

The fact that all propositional formulas can be written purely in terms of `?:`, ⊤, and ⊥ forms the basis for the *binary decision diagram*, a data structure for compactly encoding propositional formulas. Binary decision diagrams have applications in program optimization, graph algorithms, and computational complexity theory. Take CS166 or CS243 for more info!

Problem Three: Executable Logic

There's a great quote from Douglas Adams about programming a computer:

> "[I]f you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your mind. [… T]hat's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've learned something about it yourself."

To help you get a better feeling for what first-order logic formulas mean – and, importantly, what they *don't* mean – we'd like you to write a series of short programs that determine whether particular first-order logic formulas are true about specific worlds. Visit the CS103 website and download the starter project for Problem Set Two. Open `ExecutableLogic.cpp`, where you'll find six function stubs. The first-order formulas in this problem deal with sets of people who may or may not be happy and who may or may not love one another. Each person is represented as an object of type `Person`, and we've provided the following predicates functions to you:

```
bool happy(Person p)
bool loves(Person p1, Person p2)
```

These functions are written in lower-case, since that's the established C++ convention. The starter files contain a program that visualizes different groups of people and shows how each of your functions evaluates. We strongly recommend using this to check your work as you go.

i. Consider the following first-order logic formula, where P is a set of people:

$$\exists x \in P.\ Happy(x)$$

Write C++ code for a function

```
bool isFormulaTrueFor_partI(std::set<Person> P)
```

that accepts as input a set of people P and returns whether the above formula is true for that particular set of people.

ii. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ Happy(x)$$

iii. Repeat the above exercise with this first-order logic formula:

$$\exists x \in P.\ (Happy(x) \wedge Loves(x, x))$$

iv. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ (Happy(x) \rightarrow Loves(x, x))$$

v. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ (Happy(x) \rightarrow \exists y \in P.\ (Happy(y) \wedge \neg Loves(x, y)))$$

It's a lot easier to write code for this one if you use a helper function.

vi. Repeat the above exercise with this first-order logic formula:

$$\exists x \in P.\ (Happy(x) \leftrightarrow \forall y \in P.\ (Loves(x, y)))$$

As in Problem Set One, you're welcome to submit your answers to this question as many times as you'd like. To submit your work for this problem, upload the file `ExecutableLogic.cpp`.

Problem Four: First-Order Negations

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

For each of the first-order logic formulas below, find a first-order logic formula that is the negation of the original statement. Your final formula must not have any negations in it except for direct negations of predicates. For example, the negation of the formula $\forall x.\ (P(x) \to \exists y.\ (Q(x) \wedge R(y)))$ could be found by pushing the negation in from the outside inward as follows:

$$\neg(\forall x.\ (P(x) \to \exists y.\ (Q(x) \wedge R(y))))$$
$$\exists x.\ \neg(P(x) \to \exists y.\ (Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \neg\exists y.\ (Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \forall y.\ \neg(Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \forall y.\ (Q(x) \to \neg R(y)))$$

Show every step of the process of pushing the negation into the formula (along the lines of what is done above), and *please preserve the indentation from the original formula* as you go. You don't need to formally prove that your negations are correct.

We strongly recommend reading over the Guide to Negations before starting this problem.

i. $\exists p.\ (\mathit{Problem}(p) \wedge$
   $\forall g.\ (\mathit{Program}(g) \to \neg\mathit{Solves}(g, p))$
   $)$

ii. $\forall x \in \mathbb{R}.\ \forall y \in \mathbb{R}.\ (x < y \to$
   $\exists q \in \mathbb{Q}.\ (x < q \wedge q < y)$
   $)$

iii. $(\forall x.\ \forall y.\ \forall z.\ (R(x, y) \wedge R(y, z) \to R(x, z))) \quad \to \quad (\forall x.\ \forall y.\ \forall z.\ (R(y, x) \wedge R(z, y) \to R(z, x)))$

iv. $\forall x.\ \exists S.\ (\mathit{Set}(S) \wedge$
   $\forall z.\ (z \in S \leftrightarrow z = x)$
   $)$

v. $\forall k.\ (\mathit{SixClique}(k) \to$
   $\exists t.\ (\mathit{Triangle}(t, k) \wedge$
   $(\forall e.\ (\mathit{Edge}(e, t) \to \mathit{Red}(e)) \vee \forall e.\ (\mathit{Edge}(e, t) \to \mathit{Blue}(e)))$
   $)$
   $)$

Problem Five: Vacuous Truths

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

A statement is called ***vacuously true*** if either

- it's an implication of the form $P \to Q$ where P is false,
- it's a universally-quantified implication $\forall x.\ (P(x) \to Q(x))$ where $P(x)$ is never true, or
- it's a universally-quantified implication $\forall x \in S.\ Q(x)$ where S is the empty set.

These statements are true *by definition*. In the first case, the truth table for the $\to$ connective says the implication is true when the antecedent is false, and in the second and third cases we just define these sorts of statements to be true.

You might be wondering why exactly this is the case.

i. Negate the propositional formula $P \to Q$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why when $P \to Q$ is vacuously true, its negation is false.

ii. Negate the first-order formula $\forall x.\ (P(x) \to Q(x))$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why if the original formula is vacuously true, then its negation is false.

iii. Negate the first-order formula $\forall x \in S.\ Q(x)$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why if the original formula is vacuously true, then its negation is false.

Your answers to parts (i), (ii), and (iii) of this problem give another justification for vacuous truths. The three classes of statements given above have false negations in the indicated cases, and therefore to keep everything consistent we choose to define them as true.

iv. Now, look back over the code you wrote for parts (ii), (iv), and (v) of the Executable Logic problem. In the course of writing those functions, you should not have needed to handle vacuous truths by adding in extra code to specifically check whether the group in question is empty. (If you did, see if you can go back and remove it!) Briefly explain why it's not necessary to handle these cases explicitly and why your code will correctly handle empty inputs without singling this case out.

Your answer to part (iv) of this problem gives a different intuition for why statements would be vacuously true – vacuous truth naturally follows from how we might think about checking whether a universally-quantified formula would be true.

Problem Six: This, But Not That

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

Below is a series of pairs of statements about a group of people. For each pair, come up with a single group of people where the first statement is true about that group of people and the second statement is false about that group of people. To submit your answers, edit the files `ThisButNotThatI.people`, `ThisButNotThatII.people`, etc. in the `res/` directory of the starter files for this assignment with a description of those groups. There's a description in each of those files of how to specify a group of people.

The starter file `ThisButNotThat.cpp` we've provided you contain stubs of ten functions representing the ten statements here (the five "this" statements and the five "that" statements). You may ***optionally*** implement those functions in the course of solving this problem so that you can test your worlds, but you are not required to do so. You'll just be graded on the groups you submit.

	Make this statement true…	***… and this statement false.***
i.	$\forall y \in P.\ \exists x \in P.\ Loves(x, y)$	$\exists x \in P.\ \forall y \in P.\ Loves(x, y)$
ii.	$\forall x \in P.\ (Happy(x) \lor Loves(x, x))$	$(\forall x \in P.\ Happy(x)) \lor (\forall x \in P.\ Loves(x, x))$
iii.	$(\exists x \in P.\ Happy(x)) \land (\exists x \in P.\ Loves(x, x))$	$\exists x \in P.\ (Happy(x) \land Loves(x, x))$
iv.	$(\forall x \in P.\ Happy(x)) \to (\forall y \in P.\ Loves(y, y))$	$\forall x \in P.\ \forall y \in P.\ (Happy(x) \to Loves(y, y))$
v.	$\exists x \in P.\ (Loves(x, x) \to \forall y \in P.\ (Loves(y, y))$)	$(\forall x \in P.\ Loves(x, x)) \lor (\forall x \in P.\ \neg Loves(x, x))$

As a hint, if you want to make a statement false, make its negation true.

To submit your answer, upload the five `.people` files you edited in the course of solving this problem to GradeScope, and don't forget to also include the `ExecutableLogic.cpp` file from earlier in this problem set!

Problem Seven: Translating into Logic

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to First-Order Translations, which covers all the skills you'll need.)

In each of the following, write a statement in first-order logic that expresses the indicated sentence. Your statement may use any first-order construct (equality, connectives, quantifiers, etc.), but you ***must*** only use the predicates, functions, and constants provided. You do not need to provide the simplest formula possible, though we'd appreciate it if you made an effort to do so. ☺ We ***highly*** recommend reading the Guide to First-Order Logic Translations before starting this problem.

i. Given the predicate

Natural(x), which states that x is an natural number

and the functions

$x + y$, which represents the sum of x and y, and
$x \cdot y$, which represents the product of x and y

write a statement in first-order logic that says "for any $n \in \mathbb{N}$, n is even if and only if n^2 is even."

Try translating this statement assuming you have a predicate Even(x). Then, rewrite your solution without using Even(x). Numbers aren't a part of FOL, so you can't use the number 2 in your solution.

ii. Given the predicates

Person(p), which states that p is a person;
Kitten(k), which states that k is a kitten; and
HasPet(o, p), which states that o has p as a pet,

write an FOL statement that says "someone has exactly two pet kittens and no other pets."

Make sure your formula requires that the person have exactly two pet kittens; look at the lecture example of uniqueness as a starting point. Good questions to ask – is your formula false if everyone has exactly one pet kitten? Is it false if everyone has exactly three pet kittens?

iii. The ***axiom of pairing*** is the following statement: given any two distinct objects x and y, there's a set containing x and y and nothing else. Given the predicates

$x \in y$, which states that x is an element of y, and
Set(S), which states that S is a set,

write a statement in first-order logic that expresses the axiom of pairing.

iv. Given the predicates

$x \in y$, which states that x is an element of y, and
Set(S), which states that S is a set,

write a statement in first-order logic that says "every set has a power set."

As a warm-up, solve this problem assuming you have a predicate $X \subseteq Y$ that says that X is a subset of Y. Once you have that working, see if you can solve the full version of this problem.

v. Given the predicates

Lady(x), which states that x is a lady;
Glitters(x), which states that x glitters;
SureIsGold(x, y), which states that x is sure that y is gold;
Buying(x, y), which states that x buys y; and
StairwayToHeaven(x), which states that x is a Stairway to Heaven;

write a statement in first-order logic that says "there's a lady who's sure all that glitters is gold, and she's buying a Stairway to Heaven."

Problem Eight: Hereditary Sets

Let's begin with a fun little definition:

A set S is called a ***hereditary set*** if all its elements are hereditary sets.

This definition might seem strange because it's self-referential – it defines hereditary sets in terms of other hereditary sets! However, it turns out that this is a perfectly reasonable definition to work with, and in this problem you'll explore some properties of these types of sets.

i. Given the self-referential nature of the definition of hereditary sets, it's not even clear that hereditary sets even exist at all! As a starting point, prove that there is at least one hereditary set.

ii. Prove that if S is a hereditary set, then $\wp(S)$ is also a hereditary set.

After you've written up a draft of your proofs, take a minute to read over them and apply the criteria from the Proofwriting Checklist. Here are a few specific things to watch out for:

- *If you want to prove in part (ii) that a set T is a hereditary set, you need to prove the statement "every element of T is a hereditary set." That's a universally-quantified statement. If you're proving it via a direct proof, you'll probably need to pick some arbitrary element $x \in T$, then prove that x is a hereditary set by making specific claims about the variable x. Read over your proof and make sure that (1) you've introduced a new variable to refer to some arbitrarily-chosen element of T and that (2) you're making specific claims about the variable x, rather than talking in general about how elements of T behave. You may need to introduce multiple variables in the course of your proofs.*
- *A common mistake we see people make when they're just getting started is to restate definitions in the abstract in the middle of a proof. For example, we commonly see people say something like "since $A \subseteq B$, we know that every element of A is an element of B." When you're writing a proof, you can assume that whoever is reading your proof is familiar with the definitions of relevant terms, so statements like the one here that just restate a definition aren't necessary. Instead of restating definitions, try to apply those definitions. A better sentence would be something to the effect of "Since $x \in A$ and $A \subseteq B$, we see that $x \in B$," which uses the definition to conclude something about a specific variable rather than just restating the definition.*
- *Although we've just introduced first-order logic as a tool for formalizing definitions and reasoning about mathematical structures, the convention is to **not** use first-order logic notation (connectives, quantifiers, etc.) in written proofs. In a sense, you can think of first-order logic as the stage crew in the theater piece that is a proof – it works behind the scenes to make everything come together, but it's not supposed to be in front of the audience. Make sure that you're still writing in complete sentences, that you're not using symbols like $\forall$ or $\rightarrow$ in place of words like "for any" or "therefore," etc.*

Problem Nine: Symmetric Latin Squares

A ***Latin square*** is an $n \times n$ grid filled with the numbers 1, 2, 3, …, n such that every number appears in every row and every column exactly once. For example, the following are Latin squares:

1	2	3
3	1	2
2	3	1

4	2	1	3
1	3	2	4
3	1	4	2
2	4	3	1

1	3	5	2	4
2	4	1	3	5
3	5	2	4	1
4	1	3	5	2
5	2	4	1	3

A ***symmetric Latin square*** is a Latin square that is symmetric across the main diagonal (the one from the upper-left corner to the lower-right corner). That is, the elements at positions (i, j) and (j, i) are always the same. For example:

1	2	3
2	3	1
3	1	2

4	2	3	1
2	3	1	4
3	1	4	2
1	4	2	3

1	2	3	4	5
2	4	5	3	1
3	5	2	1	4
4	3	1	5	2
5	1	4	2	3

Prove that in any $n \times n$ symmetric Latin square where n is odd, every number 1, 2, 3, …, n must appear at least once on the main diagonal.

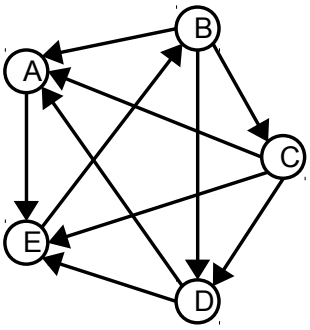
As a hint: split the Latin square into three regions – the main diagonal and the two regions above and below the main diagonal. Then think about how often each element appears in each group.

Once you've written up a draft of your proof for this problem, try out the following exercise as a way of checking your work. If you look at the sample Latin squares shown above, you can see that the result given above is not true in the case where the Latin square isn't symmetric, and it's also not true in the case where the square has even size. As a result, if your proof does not specifically use the fact that the Latin square is symmetric and does not specifically use the fact that the Latin square has odd size, it has to contain an error somewhere. Otherwise, you could change the setup to the problem and end up with a proof of an incorrect result. (Do you see why this is?) So go back over your proof and ask yourself – where, specifically, am I making reference to the fact that the Latin square is symmetric? Where, specifically, am I making reference to the fact that the Latin square has odd size? And why would my proof break down if I eliminated either of those references?

Going forward, this approach to checking your proofs – perturbing the starting assumptions and seeing where your logic breaks down – is an excellent way to smoke out any underlying logic errors.

Problem Ten: Tournament Winners

Here's one more problem to help you practice your proofwriting. It's a classic CS103 problem, and we hope that you enjoy it!



A ***tournament*** is a contest among n players. Each player plays a game against each other player, and either wins or loses the game (let's assume that there are no draws). We can visually represent a tournament by drawing a circle for each player and drawing arrows between pairs of players to indicate who won each game. For example, in the tournament to the left, player A beat player E, but lost to players B, C, and D.

A ***tournament winner*** is a player in a tournament who, for each other player, either won her game against that player, or won a game against a player who in turn won his game against that player (or both). For example, in the graph on the left, players B, C, and E are tournament winners. However, player D is ***not*** a tournament winner, because he neither beat player C, nor beat anyone who in turn beat player C. Although player D won against player E, who in turn won against player B, who then won against player C, under our definition player D is ***not*** a tournament winner. *(Make sure you understand why!)*

i. Let T be an arbitrary tournament and p be any player in that tournament. Prove the following statement: if p won more games than anyone else in T or is tied for winning the greatest number of games, then p is a tournament winner in T.

*This problem is a lot easier to solve if you draw the right picture. Something to think about: what happens if a player p **isn't** a winner in a tournament? What would that mean about player p? And finally, be careful not to make broad claims about tournaments or tournament structures without first proving them!*

A corollary of the result you proved in part (i) is that every tournament with at least one player must have at least one tournament winner – you can always pick someone who won the most games or is tied for winning the most games. However, note that you can win a tournament without necessarily winning the most games – for example, player E in the above example tournament is a winner, though player E only won a single game!

Whenever you prove a new mathematical result (in this case, every tournament with at least one player has at least one winner), it's useful to ask how "resilient" that result is. In other words, if we relax our definition of a tournament a little bit, can we still necessarily guarantee that we can always find a winner? The answer is no, and in fact, even slightly weakening the definition of a tournament invalidates this result.

Let's introduce one more definition. A ***pseudotournament*** is like a tournament, except that exactly one pair of people don't play a game against one another.

ii. Prove that for any $n \geq 2$, there's a pseudotournament P with n players and no tournament winners.

Think about the structure of what you're being asked to prove here. There's both a universal and an existential component to this statement. How do you prove an existential statement?

Optional Fun Problem: Insufficient Connectives (1 Point Extra Credit)

On some of the earlier problems on this problem set, you that every propositional logic formula could be written in terms of just the $\neg$, $\wedge$, and $\top$. You also saw that you could use just $\rightarrow$ and $\bot$, or alternatively that you could just use `?:`, $\top$, and $\bot$. Interestingly, you *cannot* express every possible propositional logic formula using just the $\leftrightarrow$ and $\bot$ connectives. Prove why not.