First-Order Translation Checklist

In this handout, we've distilled five specific points that you should check in your first-order logic statements before submitting them. They are as follows:

- ☐ ***Pair quantifiers with the appropriate connectives.***
- ☐ ***Use whitespace and indentation to clarify meaning.***
- ☐ ***Defer quantifiers until they're needed.***
- ☐ ***Check the types of all terms in the formula.***
- ☐ ***Check the scoping on each variable.***

We strongly recommend that you work through this checklist for each of the first-order statements you produced when working through Problem Set Two (and, more generally, going forward). We will specifically be looking at these details when grading your problem set.

The remainder of this handout goes into more detail about what each of these rules mean.

Pair Quantifiers With the Appropriate Connectives

In our lectures on first-order logic translations and in the Guide to Logic Translations, we stress the importance of knowing the four Aristotelian forms and how to represent each of them in first-order logic. Those forms pair the → connective with the ∀ quantifier and the ∧ connective with the ∃ quantifier. As you saw in our first lecture on first-order logic, mixing these up and pairing → with ∃ or ∧ with ∀ result in incorrect translations.

When you're reading over your translations and are getting ready to submit them, please, please, *please* do a double-check to make sure that you're pairing the quantifiers and connectives properly. It might seem silly, but mixing up the → and ∧ connectives and pairing them with the wrong quantifier is one of the single most common mistakes we make, and it's something that the TAs are extremely good at spotting. So go quantifier by quantifier through your formulas and make sure that everything agrees properly – we feel bad every time we deduct points for these sorts of errors.

For example, consider this attempted translation of the statement "there's a horse with no name:"

⚠
$$\begin{array}{l}\exists h.\ (\mathit{Horse}(h) \rightarrow \\ \quad \forall n.\ (\mathit{NameOf}(h) \neq n) \\ )\end{array}$$
⚠

The top-level quantifier here is an existential quantifier, and it's paired with the → connective, which *immediately* suggests that this formula isn't translated correctly. And indeed you can see this: imagine there's a world where every horse has a name, in which case "there's a horse with no name" is false. But this first-order logic formula would be true: just choose as h something that isn't a horse.

Similarly, consider this attempted translation of the statement "everybody has someone to lean on:"

⚠
$$\begin{array}{l}\forall p.\ (\mathit{Person}(p) \wedge \\ \quad \exists q.\ (\mathit{Person}(q) \wedge \mathit{CanLeanOn}(p, q)) \\ )\end{array}$$
⚠

Here, at the top level we see a universal quantifier paired with the ∧ connective, which *immediately* suggests that this formula isn't correct. As before, we can see this by thinking about an example. Imagine a world where everyone does indeed have someone to lean on. Now, pick as p something that isn't a person (say, the Exxon-Mobil corporation). Then $\mathit{Person}(p)$ is false, so the inside of universally-quantified formula is false for that choice of p, so the overall formula is false.

There are a few times when checking these rules can be a bit tricky. For example, let's go back to our translation of the statement "there is someone that everybody else loves," which is shown here:

$$\begin{array}{l}\exists p.\ (\mathit{Person}(p) \wedge \\ \quad \forall q.\ (\mathit{Person}(q) \wedge q \neq p \rightarrow \\ \quad\quad \mathit{Loves}(q, p) \\ \quad) \\ )\end{array}$$

If you look at the universal quantifier introducing q, you'll see that it is applied to a statement that does indeed involve the ∧ connective, which seems like it breaks the rule from before. However, remember that the ∧ connective has higher precedence than →, so the grouping here is

$$\forall q.\ ((\mathit{Person}(q) \wedge q \neq p) \rightarrow [\ldots]$$

with the implies at the top level.

As mentioned in the Guide to Logic Translations, and as you saw in lecture, you will sometimes see the ↔ connective used with the ∀ quantifier, especially in the context of set theory. That's a common context where you'll see the normal →/∀ pairing break down.

You will also sometimes see ∃ paired with ↔, which looks *really weird* the first time you see it. This often results from taking the negation of a formula that pairs ∀ with ↔. For example, consider this formula:

$$\begin{array}{l} \exists S.\ (\mathit{Set}(S) \wedge \\ \quad \forall x.\ (x \in S \leftrightarrow x = 137) \\ ) \end{array}$$

(Question to ponder: what does this say?) If we negate this formula and push the negation as deep as possible, we get this statement:

$$\begin{array}{l} \forall S.\ (\mathit{Set}(S) \rightarrow \\ \quad \exists x.\ (x \in S \leftrightarrow x \neq 137) \\ ) \end{array}$$

Notice that the innermost ∀ got flipped to an ∃, but the inner connective is still a ↔. That's totally fine. Remember that you can simplify $\neg(p \leftrightarrow q)$ to $p \leftrightarrow \neg q$, so the ↔ connective "flips" to itself when it's negated.

A great specific test you can do check if your formulas have the right quantifiers is to use the "Mount Everest" test that we did in class. Try plugging in Mount Everest in for some of the quantified variables in your formula, and see whether you accidentally make the entire formula true or the entire formula false.

Use Whitespace and Indentation to Clarify Meaning

If you're writing out a simple formula like this one:

$$\forall x \in A.\ \forall y \in A.\ (xRy \rightarrow yRx)$$

the formula can nicely and concisely fit onto a single line. But if you're writing out something more complex, it's important to use whitespace to make it cleaner and easier to read. For example, consider this formula, which says "every nonempty tournament has a tournament winner:"

$$\begin{array}{l}\forall T.\ (\mathit{Tournament}(T) \land (\exists p.\ p \in T) \rightarrow \\ \quad \exists w.\ (w \in T \land \\ \quad\quad \forall p \in T.\ (w \neq p \rightarrow \\ \quad\quad\quad (\mathit{Beat}(w, p) \lor \exists q \in T.\ (\mathit{Beat}(w, q) \land \mathit{Beat}(q, p)) \\ \quad\quad) \\ \quad) \\ )\end{array}$$

It's a good exercise to see why this works. As a hint, the general pattern is the following:

> For any tournament T that has at least one player,
> There is a player w (our winner) in T where
> For any player p other than w,
> Either w beat p or w beat some player q who in turn beat p.

This formula is lengthy and dense, but by writing it out on multiple lines and by indenting to show the nesting structure, it's significantly easier to read each piece, especially if you're comfortable with the Aristotelian forms and can recognize what each piece says.

On the other hand, imagine that you submit something like this for your TA to grade:

$$\forall T.\ (\mathit{Tournament}(T) \land (\exists p.\ p \in T) \rightarrow \exists w.\ (w \in T \land \forall p \in T.\ (w \neq p \rightarrow (\mathit{Beat}(w, p) \lor \exists q \in T.\ (\mathit{Beat}(w, q) \land \mathit{Beat}(q, p)))))$$

Here, we've crammed the entire formula onto a single line, and to do so we had to crank the font size down to 10pt. Or worse, imagine you split it across multiple lines with no predictable pattern:

$$\begin{array}{l}\forall T.\ (\mathit{Tournament}(T) \land (\exists p.\ p \in T) \rightarrow \exists w.\ (w \in T \land \forall p \in T.\ (w \neq p \rightarrow (\mathit{Beat}(w, \\ p) \lor \exists q \in T.\ (\mathit{Beat}(w, q) \land \mathit{Beat}(q, p)))))\end{array}$$

This is the mathematical equivalent of writing a program like this one:

```
int main(){for(int i=0;i<5;i++){for(int j=0;j<5;j++){cout<<(i*j)<<endl;}}return 0;}
```

This program is *technically* correct, but it's written in such a way that even a veteran programmer would have to pause and think for a bit to see what's being said. Even if this is functionally correct, it's ✓- style at best.

Just as you wouldn't write programs like the one above, you should not try to represent a complicated first-order logic formula with a number of different pieces all on a single line, and you should not put line breaks in unusual places. If you do, you make it dramatically harder to grade what you've submitted. *If you submit a formula crammed onto one line or without clear whitespace, we reserve the right to give it no points and move on*, and *if we misgrade a formula with bad arrangement because it was structured poorly, we will not regrade it!*

Defer Quantifiers Until They're Needed

In most programming languages, you have the option to declare variables at many different points in a function. It's generally considered a good idea to declare variables as late as you possibly can. For example, the following code would be considered poor style:

```
int i;
int bestIndex;
int cost;
int bestCost = -1;

for (i = 0; i < arr.length; i++) {
    cost = costOf(arr[i]);
    if (cost > bestCost) {
        bestCost = cost;
        bestIndex = i;
    }
}
```

With all the variables declared up at the top, it's hard to tell which values are transient and are only needed once per loop iteration, which values are needed just to control the loop, and which values are supposed to persist across the entire run of the loop. A much better way to write this code would be as follows:

```
int bestIndex;
int bestCost = -1;

for (int i = 0; i < arr.length; i++) {
    int cost = costOf(arr[i]);
    if (cost > bestCost) {
        bestCost = cost;
        bestIndex = i;
    }
}
```

Here, it's clearer that `bestIndex` and `bestCost` are supposed to persist across all iterations of the loop (that's why they're declared outside the loop), that `i` is the loop counter and only exists while the loop runs, and that `cost` is needed once per loop iteration and only exists during a single run of the loop. In that sense, this code is easier to read. But this code is also far less error-prone than the initial one. In the original version of the code, we could, at any point, read or write any of the four variables, making it easy to read a value that hasn't been initialized, or to read a value from a previous loop iteration that hasn't been reset yet.

In first-order logic, you have the ability to introduce variables at different points in a formula by using quantifiers. Our advice to you is the same as in programming: don't put all the quantifiers at the front of the formula. This makes the formula *much* harder to read and is very error-prone.

As an example of this, let's look at an attempted translation of the statement "there's a kitten that loves everyone." The following formula *does not* correctly represent this idea:

$$\text{⚠} \qquad \exists k.\ \forall p.\ (\mathit{Kitten}(k) \wedge \mathit{Person}(p) \to \mathit{Loves}(k, p)) \qquad \text{⚠}$$

Before we go into exactly why this doesn't work, take a minute to read over this and see if you can figure out why it's not a proper translation. The answer is on the next page.

The problem with this particular translation is that, the way the connectives are written out, this gets parsed as follows:

$$\text{⚠} \qquad \exists k.\ \forall p.\ (\ (\mathit{Kitten}(k) \wedge \mathit{Person}(p)) \rightarrow \mathit{Loves}(k, p)) \qquad \text{⚠}$$

Notice that this is essentially an implication wrapped in an existential quantifier, which is something that, as we talked about earlier, does not work out correctly! Specifically, imagine that we pick k to be something that isn't a kitten. That choice of k makes the entire formula true (do you see why?), completely independently of whether there is a kitten that loves everyone.

On the other hand, suppose we defer the quantifiers until they're actually needed. That gives us this formula, which is indeed correct:

$$\begin{array}{l} \exists k.\ (\mathit{Kitten}(k) \wedge \\ \quad \forall p.\ (\mathit{Person}(p) \rightarrow \mathit{Loves}(k, p)) \\ ) \end{array}$$

Notice here that the existential statement wraps a statement whose top-level connective is $\wedge$, which matches the general pattern we've seen so far. Specifically, picking k to be something that isn't a kitten won't make the inside of the formula true, since we need to find a choice of k where both k is a kitten and k loves every person.

As a general rule, when translating statements into first-order logic, we strongly advise against trying to guess all the quantifiers up front. Instead, follow the methodology that we describe in the Guide to Logic Translations: introduce quantifiers one at a time and only at the specific point in which you actually need them. If you do decide to put all the quantifiers up front, understand that even if your answer turns out to be right, it's still not considered good style, just as declaring all your variables up front at the top of a function can be technically correct but not good style. (Pro tip / life advice: you should not feel good about anything you do that is *technically* correct or *technically* legal.)

Check the Types of All Terms in the Formula

First-order logic draws a distinction between *objects* and *truth values*. All variables in first-order logic represent objects, and predicates applied to those functions produce truth values. You can only apply predicates or functions to objects, and you can only apply connectives or quantifiers to truth values. If you try applying a connective to an object, you end up with a syntactically invalid formula for essentially the same reason that you get an error if in a language like C or C++ you try comparing a string against an integer – the types are wrong.

Before you submit any formula, make sure that all the types work out. Start off by finding all the objects or variables in the formula. For each of them, mark them as type *object*. Then find the functions. The inputs to any function must also have type *object*, and the overall function evaluates to something of type *object*. Then find the predicates – their inputs must all be objects, and their output is of type *truth value*. Finally, look at quantifiers and propositional connectives. Their inputs must have type *truth value*, and their outputs are of type *truth value* as well.

As an example, let's imagine that we have the following predicates and functions at our disposal:

- The predicate *Puppy*(*x*), which states that *x* is a puppy.
- The predicate *Cute*(*x*), which states that *x* is cute.
- The predicate *Person*(*x*), which states that *x* is a person.
- The function *OwnerOf*(*x*), which evaluates to the owner of *x*.

Let's suppose we want to translate the statement "everyone owns at least one cute puppy" into first-order logic and that we come up with the following statement, which turns out to be incorrect:

$$\text{⚠} \quad \forall p.\ (\mathit{Person}(p) = \mathit{OwnerOf}(\exists d.\ (\mathit{Puppy}(d) \wedge \mathit{Cute}(d)))) \quad \text{⚠}$$

One way to see that this is incorrect is to notice that inside the top-level quantifier, we're comparing two terms with the equality predicate: *Person*(*p*) and *OwnerOf*(…). This, unfortunately, isn't syntactically valid, because *Person*(*p*) is a predicate and evaluates to a *truth value*, but the equality predicate can only work on *objects*. In that sense, we can immediately see that this formula can't be correct because the types don't match up properly.

There's another type error in here. Notice that we apply the *OwnerOf* predicate to an existentially-quantified statement (∃*d*. …). Remember that any quantified statement evaluates to a *truth value* (in this case, there either is a cute puppy or there isn't), and functions can only be applied to *objects*. Again, this by itself is sufficient for us to see that this formula cannot be correct.

Here's another formula that's an incorrect way of translating the statement:

$$\text{⚠} \quad \begin{array}{l} \forall p.\ (\mathit{Person}(p) \rightarrow \\ \quad \exists d.\ (\mathit{OwnerOf}(\mathit{Cute}(\mathit{Puppy}(d))) = p) \\ ) \end{array} \quad \text{⚠}$$

To see where the issue is, let's look at the expression

$$\text{⚠} \quad \mathit{OwnerOf}(\mathit{Cute}(\mathit{Puppy}(d))) = p \quad \text{⚠}$$

Let's work from the inside out. The variable *d* has type *object*, since it's a quantified variable. We're providing it as input to the predicate *Puppy*, which takes in an object and produces a truth value. But then we try to apply the predicate *Cute*, which is supposed to take in something of type *object*, to *Puppy*(*d*), which is a truth value. As a result, this formula isn't syntactically valid.

Be sure to "type-check" your formulas before submitting them. Otherwise, you risk turning in a formula that "doesn't compile," which would be a shame.

Check the Scoping on Each Variable

The variable introduced by a quantifier has a scope, and it's important to make sure that any time you reference a variable that that variable is in scope. (This is something you're hopefully familiar with from your programming experience.)

There are two general classes of scoping mistakes we see people make. The first one is to omit an important pair of parentheses. For example, suppose you want to translate the statement "there is a movie featuring both Emma Stone and Ryan Gosling." Here's an incorrect translation:

⚠ $\exists m.\ Movie(m) \wedge AppearsIn(EmmaStone, m) \wedge AppearsIn(RyanGosling, m)$ ⚠

The problem with this statement has to do with the precedence of the existential quantifier. Remember that existential quantifiers have high operator precedence – higher than anything else, in fact – so this expression is interpreted as

⚠ $(\exists m.\ Movie(m)) \wedge AppearsIn(EmmaStone, \underline{\boldsymbol{m}}) \wedge AppearsIn(RyanGosling, \underline{\boldsymbol{m}})$ ⚠

and the underlined appearances of m refer to a nonexistent variable m. It's the programming equivalent of writing the following:

```
{
    Movie m;
}
AppearsIn(EmmaStone, m);   // Error! m is out of scope.
AppearsIn(RyanGosling, m); // Error! m is out of scope.
```

This sort of error, fortunately, is easily corrected. As a rule of thumb, any time you introduce a quantifier, surround the expression you want to quantify with parentheses, as shown here:

$$\exists m.\ (Movie(m) \wedge AppearsIn(EmmaStone, m) \wedge AppearsIn(RyanGosling, m))$$

As you're reviewing all of your first-order logic formulas, make sure that every time you reference a quantified variable that you're inside of some parentheses preceded by the quantifier. If you follow our indentation guidelines from earlier (which we strongly recommend!), you can see whether you've done this right by making sure that you put in some parentheses right after the quantifier and then just look at the indentation.

The other scoping error we typically see involves quantifying over a variable at the wrong time. For example, suppose we want to translate the statement "every person has a puppy" into first-order logic. Here's an incorrect way to do this:

⚠ $\exists d.\ (Puppy(d)) \ \wedge \ \forall p.\ (Person(p) \rightarrow HasPet(p, d))$ ⚠

Notice that when we write $HasPet(p, d)$ that the variable p is in scope, but the variable d is completely out of scope. As a result, this formula isn't syntactically valid.

This type of mistake is harder to correct. The issue here is that the quantifier introducing d is in the wrong place – it's at the top-level, meaning that there's no connection between p and d. A better translation would be something like this:

$$\begin{array}{l} \forall p.\ (Person(p) \rightarrow \\ \quad \exists d.\ (Puppy(d) \wedge HasPet(p, d)) \\ ) \end{array}$$

Here, the quantifier introducing the puppy d is nested inside of the statement quantified over by p, so everything is scoped properly. You can think of this rule, in some sense, as a consequence of the advice to delay quantifiers until they're needed. We don't need to be talking about puppies until we're talking about a specific person, so we won't introduce that quantifier at the start of the formula.