Going beyond propositional logic

Consider the following statements:

p: Ling took CS245

q: Ling passed CS245

r: Ling failed CS245

Taken literally, these are all atomic statements, and formally they have no relationship with each other. But we know the following:

If Ling took CS245, and Ling did not pass CS245, then Ling failed CS245.

So we could add $p \wedge \neg q \rightarrow r$ to our system.

But what about Lucy, Larry, Lance, Lisa, ...?

We could write

p_{Lucy}: Lucy took CS245

q_{Lucy}: Lucy passed CS245

r_{Lucy}: Lucy failed CS245

p_{Larry}: Larry took CS245

q_{Larry}: Larry passed CS245

r_{Larry}: Larry failed CS245

etc.

And then we have $p_{\text{Lucy}} \wedge \neg q_{\text{Lucy}} \rightarrow r_{\text{Lucy}}$,

$p_{\text{Larry}} \wedge \neg q_{\text{Larry}} \rightarrow r_{\text{Larry}}$, etc.

So for each student s, we will need the premise $p_s \wedge \neg q_s \longrightarrow r_s$.

We may need other premises as well, e.g., $\neg(q_s \wedge r_s)$.

How many premises is that?

Do we even know how many students there are? What if there are infinitely many? How many atoms must we invent??

How can we economize the presentation?

Let $P(*)$ denote the statement, "$*$ took CS245."

Let $Q(*)$ denote the statement, "$*$ passed CS245."

Let $R(*)$ denote the statement, "$*$ failed CS245."

P, Q, and R may be viewed as functions taking some value $*$ and returning something that functions as an atom. We call them **predicate symbols**.

Hence, for example, $P(\text{Lucy})$ denotes the atomic statement, "Lucy took CS245."

Then our premises become
$$P(\text{Lucy}) \wedge \neg Q(\text{Lucy}) \rightarrow R(\text{Lucy}),$$
$$P(\text{Larry}) \wedge \neg Q(\text{Larry}) \rightarrow R(\text{Larry}), \text{ etc.}$$

4

We still haven't managed to reduce the number of premises we must adopt in order to accommodate reasoning about passing and failing into our system.

Trivially, we could reduce all of the premises to one, as follows:

$$(P(\mathrm{Lucy}) \wedge \neg Q(\mathrm{Lucy}) \to R(\mathrm{Lucy})) \wedge (P(\mathrm{Larry}) \wedge \neg Q(\mathrm{Larry}) \to R(\mathrm{Larry})) \wedge \cdots$$

But this doesn't really reduce anything.

To achieve economy, let us introduce another element of syntax into the language: the **variable**. We typically use the letters x, y, z, etc., to denote variables.

Variables act as placeholders for values. Hence, for example, if a variable x is taken to stand for Lucy, then $P(x)$ is the same as $P(\mathrm{Lucy})$. On the other hand, if we take x to mean Larry, then $P(x)$ means $P(\mathrm{Larry})$.

So we would like to say that $P(x) \wedge \neg Q(x) \rightarrow R(x)$ is a premise, no matter what value we take x to represent.

Universal statements

The statement $P(x) \land \neg Q(x) \rightarrow R(x)$ on its own doesn't really mean anything.

It says, "If x took CS245, but did not pass it, then x failed CS245."

But what is x?

We wish to say that the statement is a premise, independent of our choice of x.

For this, we introduce some additional syntax and write

$$\forall x \, (P(x) \land \neg Q(x) \rightarrow R(x)) \,,$$

which says that $P(x) \land \neg Q(x) \rightarrow R(x)$ holds **for every choice of** x.

With the universal statement

$$\forall x \, (P(x) \wedge \neg Q(x) \rightarrow R(x)) \, ,$$

as a premise, we may now say, as a consequence,

$$P(\text{Lucy}) \wedge \neg Q(\text{Lucy}) \rightarrow R(\text{Lucy}) \, ,$$

$$P(\text{Larry}) \wedge \neg Q(\text{Larry}) \rightarrow R(\text{Larry}) \, ,$$

$$P(17) \wedge \neg Q(17) \rightarrow R(17) \, ,$$

by letting x represent, respectively, Lucy, Larry, and 17.

Existential statements

What if we want to say "Someone failed CS245."? We could write

$$R(\mathrm{Lucy}) \lor R(\mathrm{Larry}) \lor \cdots$$

so long as the total number of students under consideration is finite.

Otherwise, we need to introduce a variable again. This time, we don't want to say that $R(x)$ is true, independent of our choice of x, but that there is a choice of x that makes $R(x)$ true (we may or may not know just what that choice is).

We write this as

$$\exists x \ R(x) \ ,$$

and read, "there exists an x such that $R(x)$."

Predicate logic

Ordinary propositional logic is not expressive enough to capture ideas like "every", "some", "none", etc.

When we add variables, predicate symbols, function symbols (which we haven't encountered yet), and the quantifiers $\forall$ and $\exists$ to propositional logic, we obtain what is called **predicate logic**, or **first-order logic**.

We will spend the next several slides formulating more precisely the language of predicate logic, which is considerably more complicated than its propositional counterpart.

Variables

From algebra, we take the notion of variables, single letters that are placeholders for (possibly unknown) concrete values. In computer science terms, these variables do not have types.

We follow the convention of using letters near the end of the alphabet ($u, v, w, x \ldots$) to represent variables (adding subscripts, such as x_1, if we run out).

We now need a way to represent the fact that something represented by a variable may or may not have a certain property. For this we use **predicates**, which can be thought of as Boolean functions. Any given predicate has a fixed number of arguments (this number is called the predicate's **arity**).

Unary predicates

A unary (one-argument) predicate can express an "is-a" relationship. For example, we may choose the predicate $E(\cdot)$ to express evenness. The statement "n is even" would then be translated as $E(n)$.

This statement may or may not be true, but as with propositional logic, we are delaying the precise details of the semantic interpretation of our formulas until we have completely described the proof system that manipulates them (though we will use intuition and prior knowledge to justify the language and the rules of the system).

Unary predicates need not be confined to mathematical notions; to formalize "The table is round", we might define the predicate $R(\cdot)$ to capture the idea of "is round".

Binary predicates

A binary predicate can indicate a relationship between two variables. To formalize a statement like "All students love CS 245", we might define a predicate $L(\cdot, \cdot)$. This allows us to use $L(x, y)$ in our formulas, where x represents a student and y represents a course that they might or might not love.

Note that the formula will not contain any indication of how L is to be interpreted; that is part of the semantics, to be discussed later.

There are a number of binary predicates familiar to us from math and CS, such as "greater than". Formally, we will use a slightly more awkward form such as $G(x, y)$, but informally, we may still write $x > y$, understanding that we really mean $G(x, y)$.

n-ary predicates

We can use predicates of any fixed finite arity; that is, we can define a predicate M with three arguments, and always use it with exactly three arguments, as in $M(x, y, z)$.

These are less common, but can occur in math, CS, and in English arguments; we may, for instance, wish to capture the fact that x is a buyer, y a seller, and z the agent in a real-estate transaction.

In mathematical terms, predicates can be used to define relations.

Nullary predicates

It is also possible to have predicates of arity 0; in other words, a predicate may take no arguments. For example, to represent the predicate "it is raining," we may simply use the nullary predicate symbol R, so that $R()$ denotes the statement, "It is raining."

With nullary predicates, it is customary to omit the parentheses so that we may, for example, use simply R itself to mean, "It is raining."

We see, then, that nullary predicates play the role played by atoms in propositional logic. Hence we do not need an explicit notion of atom in predicate logic; we can simply use nullary predicates.

Quantifiers

Since, intuitively, the use of a predicate is something to which a truth value may be assigned, we can connect these uses with logical connectives. $G(x, y) \wedge G(y, z)$ expresses an ordering which we usually write $x > y > z$.

But we still have not captured the idea of "every" or "all". The solution is familiar from Math 135: we introduce the quantifier $\forall$, read "for all". If S is a predicate for "is a student", and c represents CS 245, then the sentence "All students love CS 245" can be translated $\forall x (S(x) \rightarrow L(x, c))$.

The quantifier $\exists$ (read "there exists") captures "some"; using it, "Some students love CS 245" can be translated $\exists x (S(x) \wedge L(x, c))$. Note that the form is different from the previous example; why shouldn't it be $\exists x (S(x) \rightarrow L(x, c))$? This suggests that formalization can be subtle. We will spend more time on it later.

Once again, intuitively, a formula starting with a quantifier can be assigned a truth value, so we can use logical connectives. "No students love CS 245" can be translated as $\neg \exists x (S(x) \wedge L(x, c))$. Or is it $\forall x (S(x) \rightarrow \neg L(x, c))$? These are very different as formulas, but intuitively they should be semantically equivalent.

Functions

Although we could make do with the language features we have so far defined, there is one more feature added because of its enormous importance in math and CS and because it considerably simplifies many formulas. This is a way of applying functions to values.

The sentence "Tony's mother loves CS 245" can be translated as $\exists x (M(x, t) \wedge L(x, c))$. But, literally, this says "There is someone who is a mother of Tony and who loves CS 245." We know that everyone has one (biological) mother. If we use m to denote a "mother-of" function, we can use the translation $L(m(t), c)$.

As with functions in math and CS, our functions can have several arguments, but we use the convention that any given function has a fixed number of arguments.

In CS terms, functions have no types, so we can't apply logical connectives to them, though we can nest function applications – $m(m(t))$ denotes Tony's maternal grandmother. Put another way, predicates are special functions with Boolean types.

This also permits us a way to legitimize the use of constants, which we've snuck in (c for CS 245). A constant is simply a **nullary** function (one with no arguments). In this case, we don't need parentheses to enclose any arguments.

Equality

Our set of predicate symbols will always contain the **equality** predicate, formally written $= (x, y)$ but informally written $x = y$.

Later on, in our discussion of the semantics of predicate logic, we will have a way of assigning a meaning to the use of predicates. But equality is a special case; its meaning is always the same, and it will be the only predicate explicitly mentioned in our rules for valid proofs.

Predicate logic as a formal language

We have already differentiated between components of a formula to which we might assign truth values and ones to which we will not assign truth values. We formalize these as **formulas** and **terms**, respectively.

To begin with, we define the alphabet we are using, which consists of variable symbols, a chosen set $\mathcal{F}$ of function symbols, a chosen set $\mathcal{P}$ of predicate symbols, open and close parentheses, the four logical connectives, and the two quantifiers. Each function and predicate symbol has an **arity** (number of arguments) associated with it.

As with formal languages in CS 241 and CS 360, varying the alphabet by varying $\mathcal{F}$ and $\mathcal{P}$ changes the set of possible formulas.

Terms

A term is either:

- a variable, or

- $f(t_1, t_2, \ldots, t_n)$, where $n \geq 0$, f is an n-ary function symbol in $\mathcal{F}$, and $t_1, t_2, \ldots, t_n$ are terms.

In grammatical terms,

$$t ::= x \mid f(t, \ldots, t)$$

Note that nullary function applications (i.e., constants) are allowed,

in which case we allow ourselves to omit the parentheses.

Formulas

A formula is either:

- $P(t_1, t_2, \ldots t_n)$, where $n \geq 0$, P is an n-ary predicate symbol in $\mathcal{P}$, and $t_1, t_2, \ldots t_n$ are terms, or

- $\neg\phi$, where ϕ is a formula, or

- $\phi \wedge \psi$, $\phi \vee \psi$, or $\phi \rightarrow \psi$, for formulas ϕ and ψ, or

- $\forall x\phi$ or $\exists x\phi$, where ϕ is a formula and x is a variable.

In grammatical terms,

$$\phi \quad ::= \quad P(t, \ldots, t) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \forall x\phi \mid \exists x\phi \,,$$

where ϕ ranges over formulas, P ranges over predicate symbols, t ranges over terms, and x ranges over variables.

As with the language of propositional logic, the grammar we presented for predicate logic is an example of **abstract syntax**, in which we ignore details of precedence and binding, under the assumption that we already have a parse of the expression.

To override default precedence or binding priorities, we may use parentheses. We adopt the same precedence rules as before, and add that $\forall$ and $\exists$ have the same priority as $\neg$. These three bind most tightly, followed by $\wedge$ and $\vee$, followed by the right-associative $\rightarrow$.

As before, we can justify that a given string is a well-formed formula by repeated application of the recursive definitions. We can represent those applications as a parse tree. Here is the parse tree for our previous example $\exists x (M(x,t) \wedge L(x,c))$:
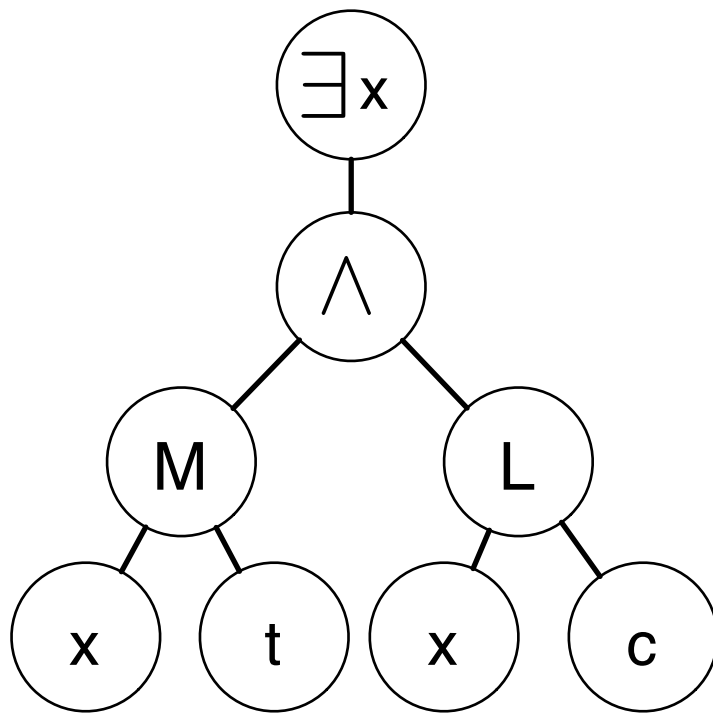
Note that the leaves of the tree are variables and constants (nullary functions).

Two classes of variables

Consider the following expression:

$$\int_2^4 (x + 2t)dx$$

Notice how x and t are treated differently. Because of the presence of the differential dx, we know that the integral is being taken with respect to x. We know what x means—it the quantity, varying between 2 and 4, over which the function is being integrated.

On the other hand, we do not know what t means—nothing in the expression tells us how to interpret t, and so we treat it as a constant throughout the evaluation, which turns out to be $6 + 4t$.

Perhaps, however, the given integral expression occurs as part of a larger expression:

$$\int_1^5 t \left(\int_2^4 (x + 2t)dx \right) dt$$

Now, by looking at the larger context, we know how to interpret t—it is the quantity, varying between 1 and 5, over which the result of the inner integral (times t) is to be integrated. The expression now evaluates to

$$\int_1^5 t(6 + 4t)dt = 3t^2 + \frac{4}{3}t^3 \Big]_1^5 = \frac{712}{3}$$

Within the smaller expression,

$$\int_2^4 (x + 2t)dx \ ,$$

however, we see a real difference between x and t. The variable x has a meaning, given to it by the differential dx—we could say that x is **bound** by the differential. The variable t does not have a specific meaning within this subexpression; we could say that t is **free**.

Two classes of variables in programs

The same phenomenon arises in computer programming. Consider the following C program:

```
int f (int x) {
    int z = 3;
    return x + y + z;
}
```

or the equivalent in Scheme:

```
(define (f x)
    (let ((z 3))
         (+ x y z))
)
```

The variables `x` (also `z`) and `y` are different in terms of what we understand about their meanings.

We know what `x` and `z` mean in `return x + y + z;`—the variable `x` refers to the integer argument that is passed to the function `f` on invocation, and the variable `z` refers to the integer declared at the beginning of the function `f`, and equal to 3. We can say that `x` and `z` are **bound** by the parameter and local variable declarations within `f`.

But we don't know what `y` means. We see no source of a definition for `y`, and unless one exists in an enclosing scope, the programs will either fail to compile or eventually crash. We say that `y` is **free** in these programs.

Free and bound variables

The same distinction between free and bound variables occurs in predicate logic formulas as well.

An occurrence of a variable x in a formula is **free** if there is no $\forall x$ or $\exists x$ on the leaf-to-root path from that occurrence in the formula's parse tree. If an occurrence is not free, it is **bound**.

Example:

$$\forall x (P(x) \wedge Q(y) \wedge R(z) \rightarrow \neg \exists y (S(y,w) \vee T(x)))$$

Here, all occurrences of x are bound, and all occurrences of w and z are free. The first occurrence of y (i.e., in $Q(y)$) is free and the second occurrence of y (i.e., in $S(y,w)$) is bound.

Formally...

Of course, we have only defined parse trees informally, so we really should use a formal recursive definition based on the recursive definition of formulas.

The set $fr(\chi)$ of occurrences of free variables in a formula χ is:

- All occurrences of all variables in χ, if χ is of the form $P(t_1, t_2, \ldots t_n)$;

- $fr(\phi)$, if χ is of the form $(\neg \phi)$;

- $fr(\phi) \cup fr(\psi)$, if χ is of the form $(\phi \wedge \psi)$, $(\phi \vee \psi)$, or $(\phi \rightarrow \psi)$;

- $fr(\phi)$ with all occurrences of x removed, if χ is of the form $(\forall x \phi)$ or $(\exists x \phi)$.

The scope of an occurrence of a quantifier $\forall x$ or $\exists x$ in a formula χ is the subtree rooted at that occurrence, minus any sub-subtrees with the same quantified variable at their roots.
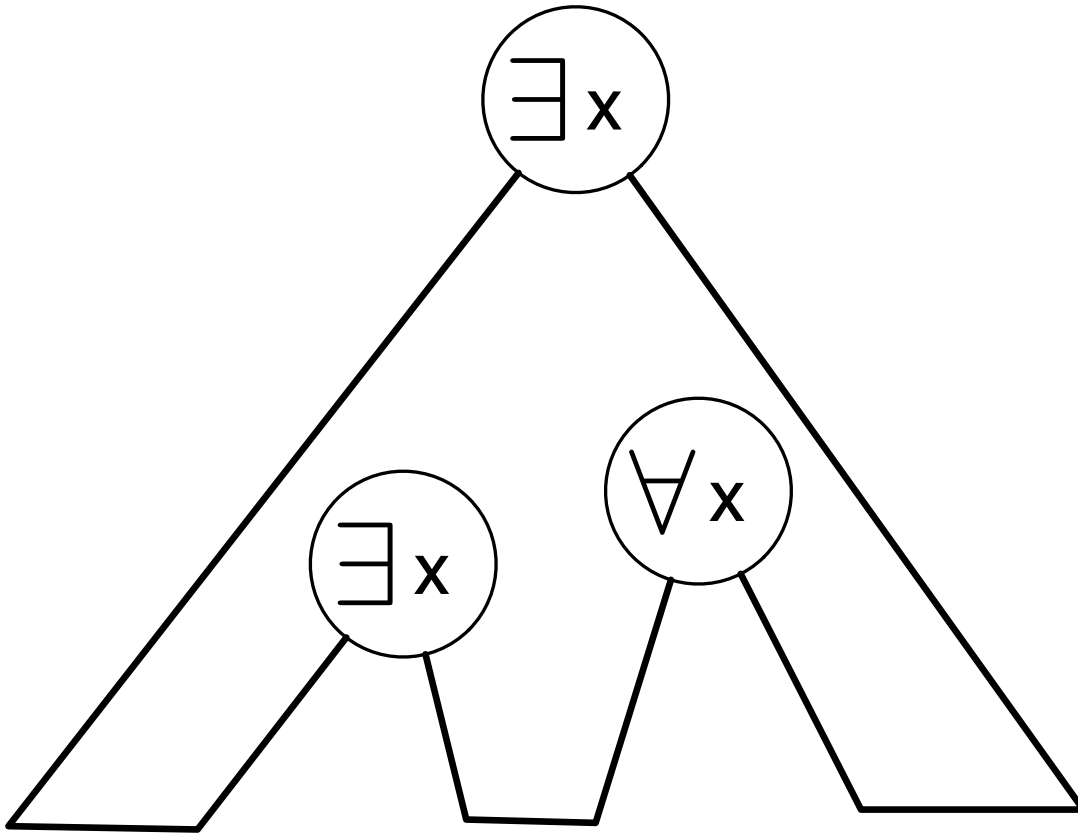
These notions may be familiar to you from lexically-scoped
programming languages. In Scheme:

```scheme
(define (f x y)
   (define (g x) (+ x y))
   (define (h y z) (* (g x) (g z)))
   (h (+ x y) (- x y)))
```

or in C-like languages:

```c
int f (int x, int y) {
    int g (int x) { return x + y; }
    int h (int y, int z) { return g(x) * g(z); }
    return h(x + y, x - y);
}
```

Here a form of "binding" occurs when identifiers are reused as parameters for locally-defined functions.

This similarity is no coincidence. These ideas were first worked out in the context of formal logic.

The lambda calculus, historically the first complete formal model of computation, was defined in order to answer questions about predicate calculus which we will soon be able to discuss.

In turn, the lambda calculus influenced the design of the programming language ALGOL, which in turn provided the syntactic basis for more recent languages such as Pascal, C/C++, and Java.

About bound variables—renaming

Because bound variables are "self-contained" in the sense that their meaning can be deduced from their binding occurrence (i.e., the $\forall$ or $\exists$ tags that introduced them), it should make no difference what names we use for them.

Example:
$$\forall x \exists y (P(x, y) \rightarrow Q(x, y))$$

We expect (though we haven't discussed semantics yet) that this formula should be equivalent to the following:

$$\forall v \exists w (P(v, w) \rightarrow Q(v, w))$$

because the universality and existentiality of the variables is unchanged; only their names are different.

The same is not true for free variables—because we do not have a supplied meaning for them, there is **no justification** for any renaming.

For example:

$$\forall x.P(x, y, z)$$

should **not** be considered equivalent to

$$\forall x.P(x, u, v)$$

and certainly not to

$$\forall x.P(x, w, w)$$

or to

$$\forall x.P(x, y, x)$$

About free variables—substitution

The proof theory of predicate logic will require us to be able to **substitute** (and also unsubstitute) terms into other terms.

Having formalized the notions of free and bound variables, we are ready to define the idea of substituting a term t into a formula ϕ. A substitution replaces all **free** occurrences of a variable x in ϕ by t, and is denoted $\phi[t/x]$.

As an example, let ϕ be $\exists x(M(x, y) \land L(x, z))$.

Then $\phi[f(w)/y]$ is $\exists x(M(x, f(w)) \land L(x, z))$.

But $\phi[f(w)/x]$ is $\exists x(M(x, y) \land L(x, z))$, because there are no free occurrences of x in ϕ.

Variable capture

Substitution looks like merely a straightforward replacement of leaves by subtrees, but we have to be careful to avoid a pitfall known as **variable capture**.

Variable capture can occur in our example
$$\phi = \exists x (M(x,y) \wedge L(x,z)).$$
$$\phi[x/y] = \exists x (M(x,x) \wedge L(x,z))$$

Here doing the substitution blindly causes a problem. The free variable y, whose meaning was supposed to be external to the expression, is now bound locally to the $\exists x$.

Had we chosen a different way of writing the formula while intuitively preserving its meaning, say $\psi = \exists w (M(w,y) \wedge L(w,z))$, then

the substitution $\psi[x/y] = \exists w(M(w,x) \wedge L(w,z))$ would not have internalized the meaning of y.

Hence, the effect of a substitution seems to be dependent upon our choice of bound variable names. This goes against our intuition, so instead we have to take steps to prevent the x that replaces y from being captured under substitution.

Said steps are precisely the bound variable renaming we exhibited above (i.e., changing x to w).

This type of renaming is intuitive for us humans, but needs more definitions in order to make it precise (and automatable).

A term t is free for a variable x in a formula ϕ if for all variables y occurring in t, no free occurrence of x is in the scope of some occurence of $\forall y$ or $\exists y$.

If t is free for x in ϕ, then the substitution $\phi[t/x]$ is safe from variable capture; otherwise, renaming of some quantifier variables and their bound occurrences is necessary before the substitution takes place.

Since we are not planning to write programs to do these manipulations, we will not formalize the rewriting process, but you should be aware that it may be necessary.

Similar issues arise in the semantics of programming languages, where the formalization is necessary.

What's coming next?

Now that we have completed the description of the language of predicate logic, we are ready to parallel our treatment of propositional logic by describing the system of natural deduction for constructing valid proofs in predicate logic.

We will then discuss the semantics of predicate logic, and say something about its completeness and soundness (we will not be able to prove these in full as we did for propositional logic), and continue with topics in predicate logic, including other proof systems, undecidability, theories, and expressivity (including some more guidelines on formalization).

Goals of this module

You should understand the language of predicate logic as a formal language with a recursive definition, and be comfortable with the terminology we use to talk about it: variables, terms, predicates, quantifiers, and formulas.

You should also understand the issues involved in the syntactic manipulation of formulas in predicate logic, and the terminology introduced to avoid possible errors: free and bound occurrences, scope, substitution, variable capture, renaming.