The language of predicate logic

Readings: Sections 2.1 and 2.2.

The language of propositional logic is not quite adequate to capture many of the mathematical arguments we make, or to describe correctness properties of programs. The language of predicate logic is another step towards this goal. Defining it, and defining the kinds of syntactic transformations we wish to make on formulas, is more complicated than it was for propositional logic.

The need for a richer language

"Every even natural number is the sum of two odd numbers."

Using propositional logic, we would have to treat this sentence as an atom. But it clearly has a compound nature. We need a language that can somehow formally express "every", "even", "is".

But both our mathematics and our programs operate on things which are not numbers (graphs, groups, strings, points, lines). Our language should be general enough to talk about properties and relationships in these cases, plus others that we may not have thought of yet.

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.

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.

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 $\forall x(S(x) \to L(x, c))$.

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 $\exists x(S(x) \to L(x, c))$? 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 $\forall x(S(x) \to \neg L(x, c))$? 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 "Ting'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 Ting 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 Ting'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
- a nullary function symbol in $\mathcal{F}$, or
- $f(t_1, t_2, \ldots, t_n)$, where f is an n-ary function symbol in $\mathcal{F}$, and $t_1, t_2, \ldots, t_n$ are terms.

In grammatical terms,

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

though in CS we may be more likely to write $\langle var \rangle$ instead of x, and so on.

Formulas

A formula is either:

- $P(t_1, t_2, \ldots t_n)$, where 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.

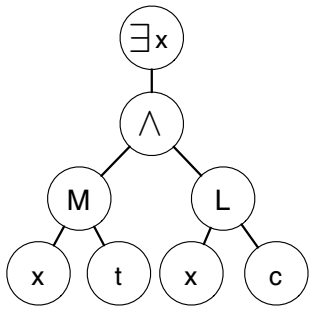
In grammatical terms,

$$\begin{aligned} \phi \quad ::= \quad & P(t, \ldots, t) \mid (\neg\phi) \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \rightarrow \psi) \\ & \mid (\forall x \phi) \mid (\exists x \phi) \end{aligned}$$

As with the language of propositional logic, we use precedence or binding priorities to allow ourselves to remove parentheses. We reuse the ones from 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 $\rightarrow$.

We will remove brackets around quantifiers on occasion, provided that the interpretation is unambiguous, but we will not specify rules for this.

As before, we 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).

Free variables and substitution

Some of the syntactic transformations in our proof system will involve the idea of substituting a term for a variable in a formula. But this has to be done carefully and consistently.

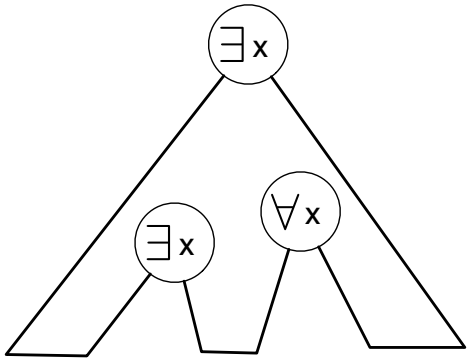
In order to define valid substitutions, we need to talk about free and bound variables. An occurrence of a variable x in a formula is **free** if there is no $\forall x$ or $\exists x$ on the leaf-to-root path from that occurrence in the formula's parse tree. If an occurrence is not free, it is **bound**.

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.

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

We are ready to define the idea of substituting a term t into a formula ϕ. The substitution replaces all free occurrences of a variable x by t, and is denoted $\phi[t/x]$.

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

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

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

This looks like a straightforward replacement of leaves by subtrees, but we have to be careful to avoid a phenomenon 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 has caused a problem. Recall that our original introduction of the predicate M was to represent "is a mother of", and $M(x,x)$ is not likely to be useful.

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 $\psi[x/y] = \exists w(M(w,x) \wedge L(w,z))$ avoids this problem.

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

A term t is free for a variable x in a formula ϕ if for all variables y occurring in ϕ, 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.