2
Predicate logic

2.1 The need for a richer language

In the first chapter, we developed propositional logic by examining it from three different angles: its proof theory (the natural deduction calculus), its syntax (the tree-like nature of formulas) and its semantics (what these formulas actually mean). From the outset, this enterprise was guided by the study of declarative sentences, statements about the world which can, for every valuation or model, be given a truth value.

We begin this second chapter by pointing out the limitations of propositional logic with respect to encoding declarative sentences. Propositional logic dealt quite satisfactorily with sentence components like *not*, *and*, *or* and *if ... then*, but the logical aspects of natural and artificial languages are much richer than that. What can we do with modifiers like *there exists ...*, *all ...*, *among ...* and *only ...*? Here, propositional logic shows clear limitations and the desire to express more subtle declarative sentences led to the design of *predicate logic*, which is also called *first-order logic*.

Let us consider the declarative sentence

Every student is younger than some instructor. (2.1)

In propositional logic, we could identify this assertion with a propositional atom p. However, that fails to reflect the finer logical structure of this sentence. What is this statement about? Well, it is about *being a student*, *being an instructor* and *being younger than somebody else*. These are all properties of some sort, so we would like to have a mechanism for expressing them together with their logical relationships and dependences.

We now use *predicates* for that purpose. For example, we could write $S(andy)$ to denote that Andy is a student and $I(paul)$ to say that Paul is an instructor. Likewise, $Y(andy, paul)$ could mean that Andy is younger than

Paul. The symbols S, I and Y are called predicates. Of course, we have to be clear about their meaning. The predicate Y could have meant that the second person is younger than the first one, so we need to specify exactly what these symbols refer to.

Having such predicates at our disposal, we still need to formalise those parts of the sentence above which speak of *every* and *some*. Obviously, this sentence refers to the individuals that make up some academic community (left implicit by the sentence), like Kansas State University or the University of Birmingham, and it says that for each student among them there is an instructor among them such that the student is younger than the instructor.

These predicates are not yet enough to allow us to express the sentence in (2.1). We don't really want to write down all instances of $S(\cdot)$ where $\cdot$ is replaced by every student's name in turn. Similarly, when trying to codify a sentence having to do with the execution of a program, it would be rather laborious to have to write down every state of the computer. Therefore, we employ the concept of a *variable*. Variables are written $u, v, w, x, y, z, \ldots$ or $x_1, y_3, u_5, \ldots$ and can be thought of as *place holders* for concrete values (like a student, or a program state). Using variables, we can now specify the meanings of S, I and Y more formally:

$$\begin{aligned} S(x): &\quad x \text{ is a student} \\ I(x): &\quad x \text{ is an instructor} \\ Y(x,y): &\quad x \text{ is younger than } y. \end{aligned}$$

Note that the names of the variables are not important, provided that we use them consistently. We can state the intended meaning of I by writing

$$I(y): \quad y \text{ is an instructor}$$

or, equivalently, by writing

$$I(z): \quad z \text{ is an instructor.}$$

Variables are mere place holders for objects. The availability of variables is still not sufficient for capturing the essence of the example sentence above. We need to convey the meaning of '**Every** *student* x *is younger than* **some** *instructor* y.' This is where we need to introduce *quantifiers* $\forall$ (read: 'for all') and $\exists$ (read: 'there exists' or 'for some') which always come attached to a variable, as in $\forall x$ ('for all x') or in $\exists z$ ('there exists z', or 'there is some z'). Now we can write the example sentence in an entirely symbolic way as

$$\forall x\,(S(x) \to (\exists y\,(I(y) \wedge Y(x,y)))).$$

Actually, this encoding is rather a paraphrase of the original sentence. In our example, the re-translation results in

> *For every x, if x is a student, then there is some y which is an instructor such that x is younger than y.*

Different predicates can have a different number of arguments. The predicates S and I have just one (they are called *unary predicates*), but predicate Y requires two arguments (it is called a *binary predicate*). Predicates with any finite number of arguments are possible in predicate logic.

Another example is the sentence

Not all birds can fly.

For that we choose the predicates B and F which have one argument expressing

$$\begin{aligned} B(x): &\quad x \text{ is a bird} \\ F(x): &\quad x \text{ can fly.} \end{aligned}$$

The sentence 'Not all birds can fly' can now be coded as

$$\neg(\forall x\,(B(x) \rightarrow F(x)))$$

saying: 'It is not the case that all things which are birds can fly.' Alternatively, we could code this as

$$\exists x\,(B(x) \wedge \neg F(x))$$

meaning: 'There is some x which is a bird and cannot fly.' Note that the first version is closer to the linguistic structure of the sentence above. These two formulas should evaluate to `T` in the world we currently live in since, for example, penguins are birds which cannot fly. Shortly, we address how such formulas can be given their meaning in general. We will also explain why formulas like the two above are indeed equivalent *semantically*.

Coding up complex facts expressed in English sentences as logical formulas in predicate logic is important – e.g. in software design with UML or in formal specification of safety-critical systems – and much more care must be taken than in the case of propositional logic. However, once this translation has been accomplished our main objective is to reason symbolically ($\vdash$) or semantically ($\models$) about the information expressed in those formulas.

In Section 2.3, we extend our natural deduction calculus of propositional logic so that it covers logical formulas of predicate logic as well. In this way we are able to prove the validity of sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in a similar way to that in the first chapter.

In Section 2.4, we generalize the valuations of Chapter 1 to a proper notion of models, real or artificial worlds in which formulas of predicate logic can be true or false, which allows us to define semantic entailment $\phi_1, \phi_2, \ldots, \phi_n \models \psi$.

The latter expresses that, given *any* such model in which all $\phi_1, \phi_2, \ldots, \phi_n$ hold, it is the case that ψ holds in that model as well. In that case, one also says that ψ is *semantically entailed* by $\phi_1, \phi_2, \ldots, \phi_n$. Although this definition of semantic entailment closely matches the one for propositional logic in Definition 1.34, the process of *evaluating a predicate formula* differs from the computation of truth values for propositional logic in the treatment of predicates (and functions). We discuss it in detail in Section 2.4.

It is outside the scope of this book to show that the natural deduction calculus for predicate logic is sound and complete with respect to semantic entailment; but it is indeed the case that

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi \quad \text{iff} \quad \phi_1, \phi_2, \ldots, \phi_n \models \psi$$

for formulas of the predicate calculus. The first proof of this was done by the mathematician K. Gödel.

What kind of reasoning must predicate logic be able to support? To get a feel for that, let us consider the following argument:

> *No books are gaseous. Dictionaries are books. Therefore, no dictionary is gaseous.*

The predicates we choose are

$$\begin{aligned} B(x): &\quad x \text{ is a book} \\ G(x): &\quad x \text{ is gaseous} \\ D(x): &\quad x \text{ is a dictionary.} \end{aligned}$$

Evidently, we need to build a proof theory and semantics that allow us to derive the validity and semantic entailment, respectively, of

$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \rightarrow B(x)) \vdash \neg\exists x\,(D(x) \wedge G(x))$$
$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \rightarrow B(x)) \models \neg\exists x\,(D(x) \wedge G(x)).$$

Verify that these sequents express the argument above in a symbolic form. Predicate logic extends propositional logic not only with quantifiers but with one more concept, that of *function symbols*. Consider the declarative sentence

> *Every child is younger than its mother.*

Using predicates, we could express this sentence as

$$\forall x \forall y \, (C(x) \wedge M(y, x) \rightarrow Y(x, y))$$

where $C(x)$ means that x is a child, $M(x, y)$ means that x is y's mother and $Y(x, y)$ means that x is younger than y. (Note that we actually used $M(y, x)$ (y is x's mother), not $M(x, y)$.) As we have coded it, the sentence says that, for all children x and any mother y of theirs, x is younger than y. It is not very elegant to say 'any of x's mothers', since we know that every individual has one and only one mother[1]. The inelegance of coding 'mother' as a predicate is even more apparent if we consider the sentence

Andy and Paul have the same maternal grandmother.

which, using 'variables' a and p for Andy and Paul and a binary predicate M for mother as before, becomes

$$\forall x \forall y \forall u \forall v \, (M(x, y) \wedge M(y, a) \wedge M(u, v) \wedge M(v, p) \rightarrow x = u).$$

This formula says that, if y and v are Andy's and Paul's mothers, respectively, and x and u are *their* mothers (i.e. Andy's and Paul's maternal grandmothers, respectively), then x and u are the same person. Notice that we used a special predicate in predicate logic, *equality*; it is a binary predicate, i.e. it takes two arguments, and is written $=$. Unlike other predicates, it is usually written in between its arguments rather than before them; that is, we write $x = y$ instead of $= (x, y)$ to say that x and y are equal.

The function symbols of predicate logic give us a way of avoiding this ugly encoding, for they allow us to represent y's mother in a more direct way. Instead of writing $M(x, y)$ to mean that x is y's mother, we simply write $m(y)$ to mean y's mother. The symbol m is a function symbol: it takes one argument and returns the mother of that argument. Using m, the two sentences above have simpler encodings than they had using M:

$$\forall x \, (C(x) \rightarrow Y(x, m(x)))$$

now expresses that every child is younger than its mother. Note that we need only one variable rather than two. Representing that Andy and Paul have the same maternal grandmother is even simpler; it is written

$$m(m(a)) = m(m(p))$$

quite directly saying that Andy's maternal grandmother is the same person as Paul's maternal grandmother.

[1] We assume that we are talking about genetic mothers, not adopted mothers, step mothers etc.

One can always do without function symbols, by using a predicate symbol instead. However, it is usually neater to use function symbols whenever possible, because we get more compact encodings. However, function symbols can be used only in situations in which we want to denote a single object. Above, we rely on the fact that every individual has a uniquely defined mother, so that we can talk about x's mother without risking any ambiguity (for example, if x had no mother, or two mothers). For this reason, we cannot have a function symbol $b(\cdot)$ for 'brother'. It might not make sense to talk about x's brother, for x might not have any brothers, or he might have several. 'Brother' must be coded as a binary predicate.

To exemplify this point further, if Mary has several brothers, then the claim that 'Ann likes Mary's brother' is ambiguous. It might be that Ann likes one of Mary's brothers, which we would write as

$$\exists x\,(B(x,m) \wedge L(a,x))$$

where B and L mean 'is brother of' and 'likes,' and a and m mean Ann and Mary. This sentence says that there exists an x which is a brother of Mary and is liked by Ann. Alternatively, if Ann likes all of Mary's brothers, we write it as

$$\forall x\,(B(x,m) \rightarrow L(a,x))$$

saying that any x which is a brother of Mary is liked by Ann. Predicates should be used if a 'function' such as 'your youngest brother' does not always have a value.

Different function symbols may take different numbers of arguments. Functions may take zero arguments and are then called *constants*: a and p above are constants for Andy and Paul, respectively. In a domain involving students and the grades they get in different courses, one might have the binary function symbol $g(\cdot,\cdot)$ taking two arguments: $g(x,y)$ refers to the grade obtained by student x in course y.

2.2 Predicate logic as a formal language

The discussion of the preceding section was intended to give an impression of how we code up sentences as formulas of predicate logic. In this section, we will be more precise about it, giving syntactic rules for the formation of predicate logic formulas. Because of the power of predicate logic, the language is much more complex than that of propositional logic.

The first thing to note is that there are two *sorts* of things involved in a predicate logic formula. The first sort denotes the objects that we are

talking about: individuals such as a and p (referring to Andy and Paul) are examples, as are variables such as x and v. Function symbols also allow us to refer to objects: thus, $m(a)$ and $g(x, y)$ are also objects. Expressions in predicate logic which denote objects are called *terms*.

The other sort of things in predicate logic denotes truth values; expressions of this kind are *formulas*: $Y(x, m(x))$ is a formula, though x and $m(x)$ are terms.

A predicate vocabulary consists of three sets: a set of predicate symbols $\mathcal{P}$, a set of function symbols $\mathcal{F}$ and a set of constant symbols $\mathcal{C}$. Each predicate symbol and each function symbol comes with an arity, the number of arguments it expects. In fact, constants can be thought of as functions which don't take any arguments (and we even drop the argument brackets) – therefore, constants live in the set $\mathcal{F}$ together with the 'true' functions which do take arguments. From now on, we will drop the set $\mathcal{C}$, since it is convenient to do so, and stipulate that constants are 0-arity, so-called *nullary*, functions.

2.2.1 Terms

The terms of our language are made up of variables, constant symbols and functions applied to those. Functions may be nested, as in $m(m(x))$ or $g(m(a), c)$: the grade obtained by Andy's mother in the course c.

Definition 2.1 Terms are defined as follows.

- Any variable is a term.
- If $c \in \mathcal{F}$ is a nullary function, then c is a term.
- If $t_1, t_2, \ldots, t_n$ are terms and $f \in \mathcal{F}$ has arity $n > 0$, then $f(t_1, t_2, \ldots, t_n)$ is a term.
- Nothing else is a term.

In Backus Naur form we may write

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

where x ranges over a set of variables var, c over nullary function symbols in $\mathcal{F}$, and f over those elements of $\mathcal{F}$ with arity $n > 0$.

It is important to note that

- the first building blocks of terms are *constants* (nullary functions) and *variables*;
- more complex terms are built from function symbols using as many previously built terms as required by such function symbols; and
- the notion of terms is dependent on the set $\mathcal{F}$. If you change it, you change the set of terms.

Example 2.2 Suppose n, f and g are function symbols, respectively nullary, unary and binary. Then $g(f(n), n)$ and $f(g(n, f(n)))$ are terms, but $g(n)$ and $f(f(n), n)$ are not (they violate the arities). Suppose $0, 1, \ldots$ are nullary, s is unary, and $+$, $-$, and $*$ are binary. Then $*(-(2, +(s(x), y)), x)$ is a term, whose parse tree is illustrated in Figure 2.14 (page 159). Usually, the binary symbols are written infix rather than prefix; thus, the term is usually written $(2 - (s(x) + y)) * x$.

2.2.2 Formulas

The choice of sets $\mathcal{P}$ and $\mathcal{F}$ for predicate and function symbols, respectively, is driven by what we intend to describe. For example, if we work on a database representing relations between our kin we might want to consider $\mathcal{P} = \{M, F, S, D\}$, referring to *being male*, *being female*, *being a son of ...* and *being a daughter of ...*. Naturally, F and M are unary predicates (they take one argument) whereas D and S are binary (taking two). Similarly, we may define $\mathcal{F} = \{mother\text{-}of, father\text{-}of\}$.

We already know what the terms over $\mathcal{F}$ are. Given that knowledge, we can now proceed to define the formulas of predicate logic.

Definition 2.3 We define the set of formulas over $(\mathcal{F}, \mathcal{P})$ inductively, using the already defined set of terms over $\mathcal{F}$:

- If $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, and if $t_1, t_2, \ldots, t_n$ are terms over $\mathcal{F}$, then $P(t_1, t_2, \ldots, t_n)$ is a formula.
- If ϕ is a formula, then so is $(\neg\phi)$.
- If ϕ and ψ are formulas, then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$ and $(\phi \to \psi)$.
- If ϕ is a formula and x is a variable, then $(\forall x\, \phi)$ and $(\exists x\, \phi)$ are formulas.
- Nothing else is a formula.

Note how the arguments given to predicates are always terms. This can also be seen in the Backus Naur form (BNF) for predicate logic:

$$\phi ::= P(t_1, t_2, \ldots, t_n) \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid (\forall x\, \phi) \mid (\exists x\, \phi) \tag{2.2}$$

where $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, t_i are terms over $\mathcal{F}$ and x is a variable. Recall that each occurrence of ϕ on the right-hand side of the $::=$ stands for any formula already constructed by these rules. (What role could predicate symbols of arity 0 play?)

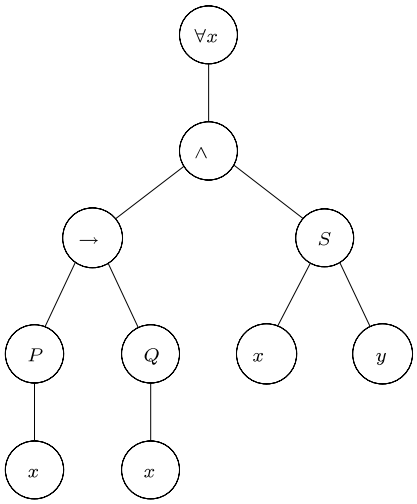

Figure 2.1. A parse tree of a predicate logic formula.

Convention 2.4 For convenience, we retain the usual binding priorities agreed upon in Convention 1.3 and add that $\forall y$ and $\exists y$ bind like $\neg$. Thus, the order is:

- $\neg$, $\forall y$ and $\exists y$ bind most tightly;
- then $\vee$ and $\wedge$;
- then $\rightarrow$, which is right-associative.

We also often omit brackets around quantifiers, provided that doing so introduces no ambiguities.

Predicate logic formulas can be represented by parse trees. For example, the parse tree in Figure 2.1 represents the formula $\forall x\,((P(x) \rightarrow Q(x)) \wedge S(x, y))$.

Example 2.5 Consider translating the sentence

Every son of my father is my brother.

into predicate logic. As before, the design choice is whether we represent 'father' as a predicate or as a function symbol.

1. As a predicate. We choose a constant m for 'me' or 'I,' so m is a term, and we choose further $\{S, F, B\}$ as the set of predicates with meanings

$$\begin{aligned} S(x,y): &\quad x \text{ is a son of } y \\ F(x,y): &\quad x \text{ is the father of } y \\ B(x,y): &\quad x \text{ is a brother of } y. \end{aligned}$$

Then the symbolic encoding of the sentence above is

$$\forall x \, \forall y \, (F(x,m) \wedge S(y,x) \rightarrow B(y,m)) \tag{2.3}$$

saying: 'For all x and all y, if x is a father of m and if y is a son of x, then y is a brother of m.'

2. As a function. We keep m, S and B as above and write f for the function which, given an argument, returns the corresponding father. Note that this works only because fathers are unique and always defined, so f really is a function as opposed to a mere relation.
   The symbolic encoding of the sentence above is now

$$\forall x \, (S(x, f(m)) \rightarrow B(x,m)) \tag{2.4}$$

meaning: 'For all x, if x is a son of the father of m, then x is a brother of m;' it is less complex because it involves only one quantifier.

Formal specifications require *domain-specific knowledge*. Domain-experts often don't make some of this knowledge explicit, so a specifier may miss important constraints for a model or implementation. For example, the specification in (2.3) and (2.4) may seem right, but what about the case when the values of x and m are equal? If the domain of kinship is not common knowledge, then a specifier may not realize that a man cannot be his own brother. Thus, (2.3) and (2.4) are not completely correct!

2.2.3 Free and bound variables

The introduction of variables and quantifiers allows us to express the notions of *all ...* and *some ...* Intuitively, to verify that $\forall x \, Q(x)$ is true amounts to replacing x by any of its possible values and checking that Q holds for each one of them. There are two important and different senses in which such formulas can be 'true.' First, if we give concrete meanings to all predicate and function symbols involved we have a *model* and can *check* whether a formula is true for this particular model. For example, if a formula encodes a required behaviour of a hardware circuit, then we would want to know whether it is true for the model of the circuit. Second, one sometimes would like to ensure that certain formulas are true *for all models*. Consider $P(c) \wedge \forall y (P(y) \rightarrow Q(y)) \rightarrow Q(c)$ for a constant c; clearly, this formula should be true no matter what model we are looking at. It is this second kind of truth which is the primary focus of Section 2.3.

Unfortunately, things are more complicated if we want to define formally what it means for a formula to be true in a given model. Ideally, we seek a definition that we could use to write a computer program verifying that a formula holds in a given model. To begin with, we need to understand that variables occur in different ways. Consider the formula

$$\forall x\,((P(x) \to Q(x)) \wedge S(x, y)).$$

We draw its parse tree in the same way as for propositional formulas, but with two additional sorts of nodes:

- The quantifiers $\forall x$ and $\exists y$ form nodes and have, like negation, just one subtree.
- Predicate expressions, which are generally of the form $P(t_1, t_2, \ldots, t_n)$, have the symbol P as a node, but now P has n many subtrees, namely the parse trees of the terms $t_1, t_2, \ldots, t_n$.

So in our particular case above we arrive at the parse tree in Figure 2.1. You can see that variables occur at two different sorts of places. First, they appear next to quantifiers $\forall$ and $\exists$ in nodes like $\forall x$ and $\exists z$; such nodes always have one subtree, subsuming their scope to which the respective quantifier applies.

The other sort of occurrence of variables is *leaf nodes containing variables.* If variables are leaf nodes, then they stand for values that still have to be made concrete. There are two principal such occurrences:

1. In our example in Figure 2.1, we have three leaf nodes x. If we walk up the tree beginning at any one of these x leaves, we run into the quantifier $\forall x$. This means that those occurrences of x are actually *bound* to $\forall x$ so they represent, or stand for, *any possible value of* x.
2. In walking upwards, the only quantifier that the leaf node y runs into is $\forall x$ but that x has nothing to do with y; x and y are different place holders. So y is *free* in this formula. This means that its value has to be specified by some additional information, for example, the contents of a location in memory.

Definition 2.6 Let ϕ be a formula in predicate logic. An occurrence of x in ϕ is free in ϕ if it is a leaf node in the parse tree of ϕ such that there is no path upwards from that node x to a node $\forall x$ or $\exists x$. Otherwise, that occurrence of x is called bound. For $\forall x\,\phi$, or $\exists x\,\phi$, we say that ϕ – minus any of ϕ's subformulas $\exists x\,\psi$, or $\forall x\,\psi$ – is the scope of $\forall x$, respectively $\exists x$.

Thus, if x occurs in ϕ, then it is bound if, and only if, it is in the scope of some $\exists x$ or some $\forall x$; otherwise it is free. In terms of parse trees, the scope of a quantifier is just its subtree, minus any subtrees which re-introduce a

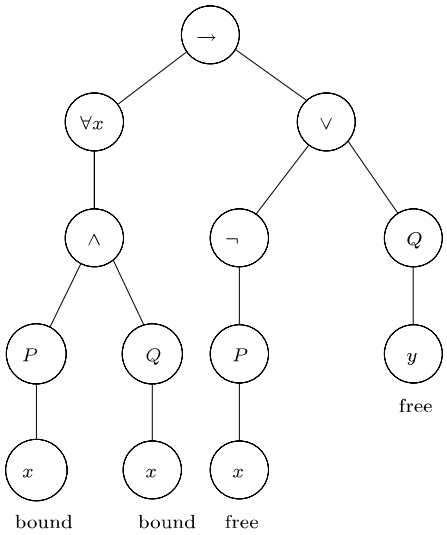

Figure 2.2. A parse tree of a predicate logic formula illustrating free and bound occurrences of variables.

quantifier for x; e.g. the scope of $\forall x$ in $\forall x\,(P(x) \to \exists x\, Q(x))$ is $P(x)$. It is quite possible, and common, that a variable is bound and free in a formula. Consider the formula

$$(\forall x\,(P(x) \wedge Q(x))) \to (\neg P(x) \vee Q(y))$$

and its parse tree in Figure 2.2. The two x leaves in the subtree of $\forall x$ are bound since they are in the scope of $\forall x$, but the leaf x in the right subtree of $\to$ is free since it is *not* in the scope of any quantifier $\forall x$ or $\exists x$. Note, however, that a single leaf either is under the scope of a quantifier, or it isn't. Hence *individual* occurrences of variables are either free or bound, never both at the same time.

2.2.4 Substitution

Variables are place holders so we must have some means of *replacing* them with more concrete information. On the syntactic side, we often need to replace a leaf node x by the parse tree of an entire term t. Recall from the definition of formulas that any replacement of x may only be a term; it could not be a predicate expression, or a more complex formula, for x serves as a term to a predicate symbol one step higher up in the parse tree (see Definition 2.1 and the grammar in (2.2)). In substituting t for x we have to

leave untouched the *bound* leaves x since they are in the scope of some $\exists x$ or $\forall x$, i.e. they stand for *some unspecified* or *all* values respectively.

Definition 2.7 Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

Substitutions are easily understood by looking at some examples. Let f be a function symbol with two arguments and ϕ the formula with the parse tree in Figure 2.1. Then $f(x, y)$ is a term and $\phi[f(x, y)/x]$ is just ϕ again. This is true because *all* occurrences of x are bound in ϕ, so *none* of them gets substituted.

Now consider ϕ to be the formula with the parse tree in Figure 2.2. Here we have one free occurrence of x in ϕ, so we substitute the parse tree of $f(x, y)$ for that free leaf node x and obtain the parse tree in Figure 2.3. Note that the bound x leaves are unaffected by this operation. You can see that the process of substitution is straightforward, but requires that it be applied *only to the free occurrences* of the variable to be substituted.

A word on notation: in writing $\phi[t/x]$, we really mean this to be the formula *obtained* by performing the operation $[t/x]$ on ϕ. Strictly speaking, the chain of symbols $\phi[t/x]$ is *not* a logical formula, but its *result* will be a formula, provided that ϕ was one in the first place.

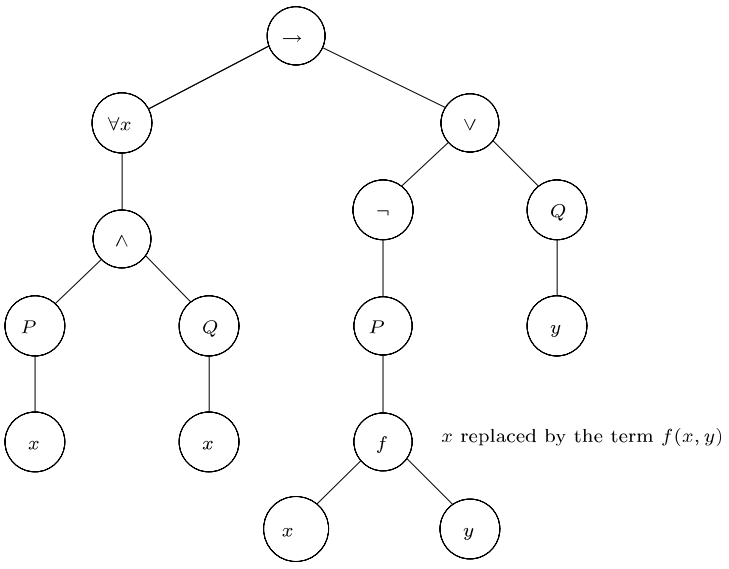

Figure 2.3. A parse tree of a formula resulting from substitution.

Unfortunately, substitutions can give rise to undesired side effects. In performing a substitution $\phi[t/x]$, the term t may contain a variable y, where free occurrences of x in ϕ are under the scope of $\exists y$ or $\forall y$ in ϕ. By carrying out this substitution $\phi[t/x]$, the value y, which might have been fixed by a concrete context, gets caught in the scope of $\exists y$ or $\forall y$. This binding capture overrides the context specification of the concrete value of y, for it will now stand for '*some unspecified*' or '*all*,' respectively. Such undesired variable captures are to be avoided at all costs.

Definition 2.8 Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

This definition is maybe hard to swallow. Let us think of it in terms of parse trees. Given the parse tree of ϕ and the parse tree of t, we can perform the substitution $[t/x]$ on ϕ to obtain the formula $\phi[t/x]$. The latter has a parse tree where all free x leaves of the parse tree of ϕ are replaced by the parse tree of t. What 't is free for x in ϕ' means is that the variable leaves of the parse tree of t won't become bound if placed into the bigger parse tree of $\phi[t/x]$. For example, if we consider x, t and ϕ in Figure 2.3, then t is free for x in ϕ since the *new* leaf variables x and y of t are not under the scope of any quantifiers involving x or y.

Example 2.9 Consider the ϕ with parse tree in Figure 2.4 and let t be $f(y, y)$. All two occurrences of x in ϕ are free. The leftmost occurrence of x could be substituted since it is not in the scope of any quantifier, but substituting the rightmost x leaf introduces a new variable y in t which becomes bound by $\forall y$. Therefore, $f(y, y)$ is not free for x in ϕ.

What if there are no free occurrences of x in ϕ? Inspecting the definition of 't is free for x in ϕ,' we see that *every* term t is free for x in ϕ in that case, since no free variable x of ϕ is below some quantifier in the parse tree of ϕ. So the problematic situation of variable capture in performing $\phi[t/x]$ cannot occur. Of course, in that case $\phi[t/x]$ is just ϕ again.

It might be helpful to compare 't is free for x in ϕ' with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.

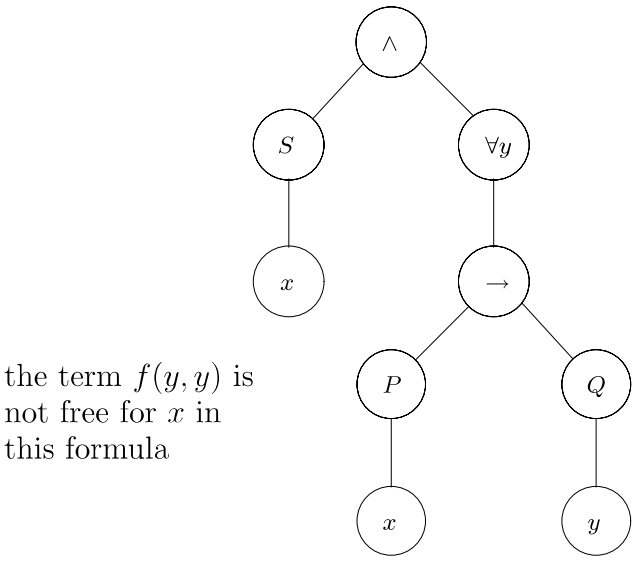

Figure 2.4. A parse tree for which a substitution has dire consequences.

2.3 Proof theory of predicate logic

2.3.1 Natural deduction rules

Proofs in the natural deduction calculus for predicate logic are similar to those for propositional logic in Chapter 1, except that we have new proof rules for dealing with the quantifiers and with the equality symbol. Strictly speaking, we are *overloading* the previously established proof rules for the propositional connectives $\wedge$, $\vee$ etc. That simply means that any proof rule of Chapter 1 is still valid for logical formulas of predicate logic (we originally defined those rules for logical formulas of propositional logic). As in the natural deduction calculus for propositional logic, the additional rules for the quantifiers and equality will come in two flavours: introduction and elimination rules.

The proof rules for equality First, let us state the proof rules for equality. Here equality does not mean syntactic, or intensional, equality, but equality in terms of computation results. In either of these senses, any term t has to be equal to itself. This is expressed by the introduction rule for equality:

$$\frac{}{t = t}\,{=}\mathrm{i} \tag{2.5}$$

which is an axiom (as it does not depend on any premises). Notice that it