CHAPTER 9

Herbrand Logic

9.1 Introduction

Relational Logic, as defined in Chapter 6, allows us to axiomatize worlds with varying numbers of objects. The main restriction is that the worlds must be finite (since we have only finitely many constants to refer to these objects).

Often, we want to describe worlds with infinitely many objects. For example, it would be nice to axiomatize arithmetic over the integers or to talk about sequences of objects of varying lengths. Unfortunately, this is not possible due to the finiteness restriction of Relational Logic.

One way to get infinitely many terms is to allow our vocabulary to have infinitely many object constants. While there is nothing wrong with this in principle, it makes the job of axiomatizing things effectively impossible, as we would have to write out infinitely many sentences in many cases.

In this chapter, we explore an alternative to Relational Logic, called Herbrand Logic, in which we can name infinitely many objects with a finite vocabulary. The trick is to expand our language to include not just object constants but also complex terms that can be built from object constants in infinitely many ways. By constructing terms in this way, we can get infinitely many names for objects; and, because our vocabulary is still finite, we can finitely axiomatize some things in a way that would not be possible with infinitely many object constants.

In this chapter, we proceed through the same stages as in the introduction to Relational Logic. We start with syntax and semantics. We then discuss evaluation and satisfaction. We look at some examples. And we conclude with a discussion of some of the properties of Herbrand Logic.

9.2 Syntax and Semantics

The syntax of Herbrand Logic is the same as that of Relational Logic except for the addition of *function constants* and *functional expressions*.

As we shall see, function constants are similar to relation constants in that they are used in forming complex expressions by combining them with an appropriate number of arguments. Accordingly, each function constant has an associated *arity*, i.e. the number of arguments with which that function constant can be combined. A function constant that can combined with a single argument is said to be *unary*; one that can be combined with two arguments is said to be *binary*; one that can be combined with three arguments is said to be *ternary*; more generally, a function constant that can be combined with n arguments is said to be n-ary.

A *functional expression*, or *functional term*, is an expression formed from an n-ary function constant and n terms enclosed in parentheses and separated by commas. For example, if f is a binary function constant and if a and y are terms, then $f(a,y)$ is a functional expression, as are $f(a,a)$ and $f(y,y)$.

Note that, unlike relational sentences, functional expressions can be nested within other functional expressions. For example, if g is a unary function constant and if a is a term, $g(a)$ and $g(g(a))$ and $g(g(g(a)))$ are all functional expressions.

Finally, in Herbrand Logic, we define a *term* to be a variable or an object constant or a functional expression. The definition here is the same as before except for the addition of functional expressions to this list of possibilities.

And that is all. Relational sentences, logical sentences, and quantified sentences are defined exactly as for ordinary Relational Logic. The only difference between the two languages is that Herbrand Logic allows for functional expressions inside of sentences whereas ordinary Relational Logic does not.

The semantics of Herbrand Logic is effectively the same as that of Relational Logic. The key difference is that, in the presence of functions, the Herbrand base for such a language is infinitely large.

As before, we define the *Herbrand base* for a vocabulary to be the set of all ground relational sentences that can be formed from the constants of the language. Said another way, it is the set of all sentences of the form $r(t_1,...,t_n)$, where r is an n-ary relation constant and t_1, ... , t_n are ground terms.

For a vocabulary with a single object constant a and a single unary function constant f and a single unary relation constant r, the Herbrand base consists of the sentences shown below.

$$\{r(a), r(f(a)), r(f(f(a))), ...\}$$

A *truth assignment* for Herbrand Logic is a mapping that gives each ground relational sentence in the Herbrand base a unique truth value. This is the same as for Relational Logic. The main difference from Relational Logic is that, in this case, a truth assignment is necessarily infinite, since there are infinitely many elements in the Herbrand Base.

Luckily, things are not always so bad. In some cases, only finitely many elements of the Herbrand base are true. In such cases, we can describe a truth assignment in finite space by writing out the elements that are true and assuming that all other elements are false. We shall see some examples of this in the coming sections.

The rules defining the truth of logical sentences in Herbrand Logic are the same as those for logical sentences in Propositional Logic and Relational Logic, and the rules for quantified sentences in Herbrand Logic are exactly the same as those for Relational Logic.

9.3 Evaluation and Satisfaction

The concept of evaluation for Herbrand Logic is the same as that for Relational Logic. Unfortunately, evaluation is usually not practical in this case for two reasons. First of all, truth assignments are infinite in size and so we cannot always write them down. Even when we can finitely characterize a truth assignment (e.g. when the set of true sentences is finite), we may not be able to evaluate quantified sentences mechanically in all cases. In the case of a universally quantified formula, we need to check all instances of the scope, and there are infinitely many possibilities. In the case of an existentially quantified sentence, we need to enumerate possibilities until we find one that succeeds, and we may never find one if the existentially quantified sentence is false.

Satisfaction has similar difficulties. The truth tables for Herbrand Logic are infinitely large and so we cannot write out or check all possibilities.

The good news is that, even though evaluation and satisfaction are not directly computable, there are effective procedures for indirectly determining validity, contingency, unsatisfiability, logical entailment, and so forth that work in many cases even when our usual direct methods fail. The key is symbolic manipulation of various sorts, e.g. the generation of proofs, which we describe in the next few chapters. But, first, in order to gain some intuitions about the power of Herbrand Logic, we look at some examples.

9.4 Example - Peano Arithmetic

Peano Arithmetic differs from Modular Arithmetic (axiomatized in Section 6.8) in that it applies to all natural numbers (0, 1, 2, 3, ...). Since there are infinitely many such numbers, we need infinitely many terms.

As mentioned in the introduction, we can get infinitely many terms by expanding our vocabulary with infinitely many object constants. Unfortunately, this makes the job of axiomatizing arithmetic effectively impossible, as we would have to write out infinitely many sentences.

An alternative approach is to represent numbers using a single object constant (e.g. 0) and a single unary function constant (e.g. *s*). We can then represent every number *n* by applying the function constant to 0 exactly *n* times. In this encoding, *s*(0) represents 1; *s*(*s*(0)) represents 2; and so forth. With this encoding, we automatically get an infinite universe of terms, and we can write axioms defining addition and multiplication as simple variations on the axioms of Modular Arithmetic.

Unfortunately, even with this representation, axiomatizing Peano Arithmetic is more challenging than axiomatizing Modular Arithmetic. We cannot just write out ground relational sentences to characterize our relations, because there are infinitely many cases to consider. For Peano Arithmetic, we must rely on logical sentences and quantified sentences, not just because they are more economical but because they are the only way we can characterize our relations in finite space.

Let's look at the same relation first. The axioms shown here define the *same* relation in terms of 0 and *s*.

$$\forall x.same(x,x)$$
$$\forall x.(\neg same(0,s(x)) \wedge \neg same(s(x),0))$$
$$\forall x.\forall y.(\neg same(x,y) \Rightarrow \neg same(s(x),s(y)))$$

It is easy to see that these axioms completely characterize the *same* relation. By the first axiom, the same relation holds of every term and itself.

$$same(0,0)$$
$$same(s(0),s(0))$$
$$same(s(s(0)),s(s(0)))$$
$$...$$

The other two axioms tell us what is not true. The second axiom tells us that 0 is not the same as any composite term. The same holds true with the arguments reversed.

$$\neg same(0,s(0)) \qquad \neg same(s(0),0)$$
$$\neg same(0,s(s(0))) \qquad \neg same(s(s(0)),0)$$
$$\neg same(0,s(s(s(0)))) \qquad \neg same(s(s(s(0))),0)$$
$$... \qquad ...$$

The third axiom builds on these results to show that non-identical composite terms of arbitrary complexity do not satisfy the same relation. Viewed the other way around, to see that two non-identical terms are not the same, we just strip away occurrences of s from each term till one of the two terms becomes 0 and the other one is not 0. By the second axiom, these are not the same, and so the original terms are not the same.

$$\neg same(s(0),s(s(0))) \qquad \neg same(s(s(0)),s(0))$$
$$\neg same(s(0),s(s(s(0)))) \qquad \neg same(s(s(s(0))),s(0))$$
$$\neg same(s(0),s(s(s(s(0))))) \qquad \neg same(s(s(s(s(0)))),s(0))$$
$$... \qquad ...$$

Once we have the *same* relation, we can define the other relations in our arithmetic. The following axioms define the plus relation in terms of 0, *s*, and *same*. Adding 0 to any number results in that number. If adding a number *x* to a number *y* produces a number *z*, then adding the successor of *x* to *y* produces the successor of *z*. Finally, we have a functionality axiom for *plus*.

$$\forall y.plus(0,y,y)$$
$$\forall x.\forall y.\forall z.(plus(x,y,z) \Rightarrow plus(s(x),y,s(z)))$$
$$\forall x.\forall y.\forall z.\forall w.(plus(x,y,z) \wedge \neg same(z,w) \Rightarrow \neg plus(x,y,w))$$

The axiomatization of multiplication is analogous. Multiplying any number by 0 produces 0. If a number *z* is the product of *x* and *y* and *w* is the sum of *y* and *z*, then *w* is the product of the successor of *x* and *y*. As before, we have a functionality axiom.

$$\forall y.times(0,y,0)$$
$$\forall x.\forall y.\forall z.\forall w.(times(x,y,z) \wedge plus(y,z,w) \Rightarrow times(s(x),y,w))$$
$$\forall x.\forall y.\forall z.\forall w.(times(x,y,z) \wedge \neg same(z,w) \Rightarrow \neg times(x,y,w))$$

That's all we need - just three axioms for *same* and three axioms for each arithmetic function.

Before we leave our discussion of Peano arithmetic, it is worthwhile to look at the concept of Diophantine equations. A *polynomial equation* is a sentence composed using only addition, multiplication, and exponentiation with fixed exponents (that is numbers not variables). For example, the expression shown below in traditional math notation is a polynomial equation.

$$x^2 + 2y = 4z$$

A *natural Diophantine equation* is a polynomial equation in which the variables are restricted to the natural numbers. For example, the polynomial equation here is also a Diophantine equation and happens to have a solution in the natural numbers, viz. x=4 and y=8 and z=8.

Diophantine equations can be readily expressed as sentences in Peano Arithmetic. For example, we can represent the Diophantine equation above with the sentence shown below.

$$\exists x.\exists y.\exists z.\forall u.\forall v.\forall w.(times(x,x,u) \wedge times(2,y,v) \wedge plus(u,v,w) \Rightarrow times(4,z,w))$$

This is a little messy, but it is doable. And we can always clean things up by adding a little syntactic sugar to our notation to make it look like traditional math notation.

Once this mapping is done, we can use the tools of logic to work with these sentences. In some cases, we can find solutions; and, in some cases, we can prove that solutions do not exist. This has practical value in some situations, but it also has significant theoretical value in establishing important properties of Herbrand Logic, a topic that we discuss in a later section.

9.5 Example - Linked Lists

A list is a finite sequence of objects. Lists can be flat, e.g. $[a, b, c]$. Lists can also be nested within other lists, e.g. $[a, [b, c], d]$.

A linked list is a way of representing nested lists of variable length and depth. Each element is represented by a cell containing a value and a pointer to the remainder of the list. Our goal in this example is to formalize linked lists and define some useful relations.

To talk about lists of arbitrary length and depth, we use the binary function constant *cons*, and we use the object constant *nil* to refer to the empty list. In particular, a term of the form $cons(\tau_1, \tau_2)$ designates a sequence in which τ_1 denotes the first element and τ_2 denotes the rest of the list. With this function constant, we can encode the list $[a, b, c]$ as follows.

$$cons(a, cons(b, cons(c, nil)))$$

The advantage of this representation is that it allows us to describe functions and relations on lists without regard to length or depth.

As an example, consider the definition of the binary relation *member*, which holds of an object and a list if the object is a top-level member of the list. Using the function constant *cons*, we can characterize the *member* relation as shown below. Obviously, an object is a member of a list if it is the first element; however, it is also a member if it is member of the rest of the list.

$$\forall x.\forall y.member(x, cons(x, y))$$
$$\forall x.\forall y.\forall z.(member(x, z) \Rightarrow member(x, cons(y, z)))$$

In similar fashion, we can define functions to manipulate lists in different ways. For example, the following axioms define a relation called *append*. The value of *append* (its last argument) is a list consisting of the elements in the list supplied as its first argument followed by the elements in the list supplied as its second. For example, we would have $append(cons(a,nil), cons(b, cons(c, nil)), cons(a, cons(b, cons(c, nil))))$. And, of course, we need negative axioms as well (omitted here).

$$\forall z.append(nil, z, z)$$
$$\forall x.\forall y.\forall z.(append(y, z, w) \Rightarrow append(cons(x, y), z, cons(x,w)))$$

We can also define relations that depend on the structure of the elements of a list. For example, the *among* relation is true of an object and a list if the object is a member of the list, if it is a member of a list that is itself a member of the list, and so on. (And, once again, we need negative axioms.)

$$\forall x.among(x, x)$$
$$\forall x.\forall y.\forall z.(among(x, y) \vee among(x, z) \Rightarrow among(x, cons(y, z)))$$

Lists are an extremely versatile representational device, and the reader is encouraged to become as familiar as possible with the techniques of writing definitions for functions and relations on lists. As is true of many tasks, practice is the best approach to gaining skill.

9.6 Example - Pseudo English

Pseudo English is a formal language that is intended to approximate the syntax of the English language. One way to define the syntax of Pseudo English is to write grammatical rules in Backus Naur Form (BNF). The rules shown below illustrate this approach for a small subset of Pseudo English. A sentence is a noun phrase followed by a verb phrase. A noun phrase is either a noun or

two nouns separated by the word *and*. A verb phrase is a verb followed by a noun phrase. A noun is either the word *Mary* or the word *Pat* or the word *Quincy*. A verb is either *like* or *likes*.

<sentence> ::= <np> <vp>
<np> ::= <noun>
<np> ::= <noun> "and" <noun>
<vp> ::= <verb> <np>
<noun> ::= "mary" | "pat" | "quincy"
<verb> ::= "like" | "likes"

Alternatively, we can use Herbrand Logic to formalize the syntax of Pseudo English. The sentences shown below express the grammar described in the BNF rules above. (We have dropped the universal quantifiers here to make the rules a little more readable.) Here, we are using the *append* relation defined in the section of lists.

$$np(x) \wedge vp(y) \wedge append(x,y,z) \Rightarrow sentence(z)$$
$$noun(x) \Rightarrow np(x)$$
$$noun(x) \wedge noun(y) \wedge append(x,\text{and},z) \wedge append(z,y,w) \Rightarrow np(w)$$
$$verb(x) \wedge np(y) \wedge append(x,y,z) \Rightarrow vp(z)$$
$$noun(mary)$$
$$noun(pat)$$
$$noun(quincy)$$
$$verb(like)$$
$$verb(likes)$$

Using these sentences, we can test whether a given sequence of words is a syntactically legal sentence in Pseudo English and we can use our logical entailment procedures to enumerate syntactically legal sentences, like those shown below.

mary likes pat
pat and quincy like mary
mary likes pat and quincy

One weakness of our BNF and the corresponding axiomatization is that there is no concern for agreement in number between subjects and verbs. Hence, with these rules, we can get the following expressions, which in Natural English are ungrammatical.

× *mary like pat*
× *pat and quincy likes mary*

Fortunately, we can fix this problem by elaborating our rules just a bit. In particular, we add an argument to some of our relations to indicate whether the expression is singular or plural. Here, 0 means singular, and 1 means plural. We then use this to block sequences of words where the numbers do not agree.

$np(x,w) \wedge vp(y,w) \wedge append(x,y,z) \Rightarrow sentence(z)$
$noun(x) \Rightarrow np(x,0)$
$noun(x) \wedge noun(y) \wedge append(x,\text{and},z) \wedge append(z,y,w) \Rightarrow np(w,1)$
$verb(x,w) \wedge np(y,v) \wedge append(x,y,z) \Rightarrow vp(z,w)$
$noun(mary)$
$noun(pat)$
$noun(quincy)$
$verb(like,1)$
$verb(likes,0)$

With these rules, the syntactically correct sentences shown above are still guaranteed to be sentences, but the ungrammatical sequences are blocked. Other grammatical features can be formalized in similar fashion, e.g. gender agreement in pronouns (*he* versus *she*), possessive adjectives (*his* versus *her*), reflexives (like *himself* and *herself*), and so forth.

9.7 Example - Metalevel Logic

Throughout this book, we have been writing sentences in English about sentences in Logic, and we have been writing informal proofs in English about formal proofs in Logic. A natural question to ask is whether it is possible formalize Logic within Logic. The answer is yes. The limits of what can be done are very interesting. In this section, we look at a small subset of this problem, viz. using Herbrand Logic to formalize information about Propositional Logic.

The first step in formalizing Propositional Logic in Herbrand Logic is to represent the syntactic components of Propositional Logic.

In what follows, we make each proposition constant in our Propositional Logic language an object constant in our Herbrand Logic formalization. For example, if our Propositional Logic language has proposition constants p, q, and r, then p, q, and r are object constants in our formalization.

Next, we introduce function constants to represent constructors of complex sentences. There is one function constant for each logical operator - *not* for $\neg$, *and* for $\wedge$, *or* for $\vee$, *if* for $\Rightarrow$, and *iff* for $\Leftrightarrow$. Using these function constants, we represent Propositional Logic sentences as terms in our language. For example, we use the term $and(p,q)$ to represent the Propositional Logic sentence $(p \wedge q)$; and we use the term $if(and(p,q),r)$ to represent the Propositional Logic sentence $(p \wedge q \Rightarrow r)$.

Finally, we introduce a selection of relation constants to express the types of various expressions in our Propositional Logic language. We use the unary relation constant *proposition* to assert that an expression is a proposition. We use the unary relation constant *negation* to assert that an expression is a negation. We use the unary relation constant *conjunction* to assert that an expression is a conjunction. We use the unary relation constant *disjunction* to assert that an expression is a disjunction. We use the unary relation constant *implication* to assert that an expression is an implication. We use the unary relation constant *biconditional* to assert that an expression is a biconditional. And we use the unary relation constant *sentence* to assert that an expression is a sentence.

With this vocabulary, we can characterize the syntax of our language as follows. We start with declarations of our proposition constants.

$proposition(p)$
$proposition(q)$
$proposition(r)$

Next, we define the types of expressions involving our various logical operators.

$$\forall x.(sentence(x) \Rightarrow negation(not(x)))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow conjunction(and(x,y)))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow disjunction(or(x,y)))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow implication(if(x,y)))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow biconditional(iff(x,y)))$$

Finally, we define sentences as expressions of these types.

$$\forall x.(proposition(x) \Rightarrow sentence(x))$$
$$\forall x.(negation(x) \Rightarrow sentence(x))$$
$$\forall x.(conjunction(x) \Rightarrow sentence(x))$$
$$\forall x.(disjunction(x) \Rightarrow sentence(x))$$
$$\forall x.(implication(x) \Rightarrow sentence(x))$$
$$\forall x.(biconditional(x) \Rightarrow sentence(x))$$

Note that these sentences constrain the types of various expressions but do not define them completely. For example, we have not said that *not*(*p*) is *not* a conjunction. It is possible to make our definitions more complete by writing negative sentences. However, they are a little messy, and we do not need them for the purposes of this section.

With a solid characterization of syntax, we can formalize our rules of inference. We start by representing each rule of inference as a relation constant. For example, we use the ternary relation constant *ai* to represent And Introduction, and we use the binary relation constant *ae* to represent And Elimination. With this vocabulary, we can define these relations as shown below.

$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow ai(x,y,and(x,y)))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow ae(and(x,y),x))$$
$$\forall x.\forall y.(sentence(x) \wedge sentence(y) \Rightarrow ae(and(x,y),y))$$

In similar fashion, we can define proofs - both linear and structured. We can even define truth assignments, satisfaction, and the properties of validity, satisfiability, and so forth. Having done all of this, we can use the proof methods discussed in the next chapters to prove our metatheorems about Propositional Logic.

We can use a similar approach to formalizing Herbrand Logic within Herbrand Logic. However, in that case, we need to be very careful. If done incorrectly, we can write paradoxical sentences, i.e. sentences that are neither true nor false. For example, a careless formalization leads to formal versions of sentences like *This sentence is false*, which is self-contradictory, i.e. it cannot be true and cannot be false. Fortunately, with care it is possible to avoid such paradoxes and thereby get useful work done.

9.8 Undecidability

The good news about Herbrand Logic is that it is highly expressive. We can formalize things in Herbrand Logic that cannot be formalized (at least in finite form) in Relational Logic. For example, we showed how to define addition and multiplication in finite form. This is not possible with Relational Logic and in other logics (e.g. First-Order Logic).

The bad news is that the questions of unsatisfiability and logical entailment for Herbrand Logic are not effectively computable. Explaining this in detail is beyond the scope of this course. However,

we can give a line of argument that suggests why it is true. The argument reduces a problem that is generally accepted to be non-semidecidable to the question of unsatisfiability / logical entailment for Herbrand Logic. If our logic were semidecidable, then this other question would be semidecidable as well; and, since it is known not to be semidecidable, then Herbrand Logic must not be semidecidable either.

As we know, Diophantine equations can be readily expressed as sentences in Herbrand Logic. For example, we can represent the solvability of Diophantine equation $3x^2=1$ with the sentence shown below.

$$\exists x.\exists y.(times(x, x, y) \wedge times(s(s(s(0))), y, s(0)))$$

We can represent every Diophantine equation in an analogous way. We can express the unsolvability of a Diophantine equation by negating the corresponding sentence. We can then ask the question of whether the axioms of arithmetic logically entail this negation or, equivalently, whether the axioms of Arithmetic together with the unnegated sentence are unsatisfiable.

The problem is that it is well known that determining whether Diophantine equations are unsolvable is not semidecidable. If we could determine the unsatisfiability of our encoding of a Diophantine equation, we could decide whether it is unsolvable, contradicting the non-semidecidability of that problem.

Note that this does not mean Herbrand Logic is useless. In fact, it is great for expressing such information; and we can prove many results, even though, in general, we cannot prove everything that follows from arbitrary sets of sentences in Herbrand Logic. We discuss this issue further in later chapters.

Recap

Herbrand Logic is an extended version of Relational Logic that includes *functional expressions*. Since functional expressions can be composed with each other in infinitely many ways, the Herbrand base for Herbrand Logic is infinite, allowing us to axiomatize infinite relations with a finite vocabulary. Other than the addition of functional expressions, the syntax and semantics of Herbrand Logic is the same as that of Relational Logic. Questions of unsatisfiability and logical entailment can sometimes be computed in Herbrand Logic, though in general those questions are not effectively computable.

Exercises

Exercise 9.1: Say whether each of the following expressions is a syntactically legal sentence of Herbrand Logic. Assume that a and b are object constants, f is a unary function constant, and p is a unary relation constant.

(a) $p(a)$
(b) $p(f(a))$
(c) $f(f(a))$
(d) $p(f(f(a)))$
(e) $p(f(p(a)))$

Exercise 9.2: Say whether each of the following sentences is logically entailed by the sentences in Section 9.4.

(*a*) *same*(*s*(0),*s*(*s*(*s*(0))))
(*b*) *plus*(*s*(*s*(0)),*s*(*s*(*s*(0))),*s*(*s*(*s*(*s*(*s*(0))))))
(*c*) *times*(*s*(*s*(0)),*s*(*s*(*s*(0))),*s*(*s*(*s*(*s*(*s*(0))))))
(*d*) *times*(*s*(0),*s*(*s*(*s*(0))),*s*(*s*(*s*(0))))

Exercise 9.3: Say whether each of the following sentences is logically entailed by the sentences in Section 9.5.

(*a*) *append*(*nil*, *nil*, *nil*)
(*b*) *append*(*cons*(*a*, *nil*), *nil*, *cons*(*a*, *nil*))
(*c*) *append*(*cons*(*a*, *nil*), *cons*(*b*, *nil*), *cons*(*a*, *b*))
(*d*) *append*(*cons*(*cons*(*a*, *nil*), *nil*), *cons*(*b*, *nil*), *cons*(*a*, *cons*(*b*, *nil*)))

Exercise 9.4: Say whether each of the following sentences is a grammatical sentence of Pseudo English according to the enhanced grammar presented at the end of Section 9.6.

(*a*) *Mary likes Pat and Quincy*.
(*b*) *Mary likes Pat and Mary likes Quincy*.
(*c*) *Mary likes Mary*.
(*d*) *Mary likes herself*.

Exercise 9.5: Say whether each of the following sentences is logically entailed by the sentences in Section 9.7.

(*a*) *conjunction*(*and*(*not*(*p*), *not*(*q*)))
(*b*) *conjunction*(*not*(*or*(*not*(*p*), *not*(*q*))))
(*c*) *ae*(*and*(*p*, *or*(*p*, *q*)), *or*(*p*, *q*))
(*d*) *ae*(*and*(*p*, *or*(*p*, *q*)), *and*(*p*, *q*))