5
Mathematical Background

In this chapter we present some background in mathematical logic in the context of software analysis. This material may be review for some readers, but we encourage all to at least skim this chapter to gain understanding of our notation, terminology, and use in writing SPARK programs. You may wish to consult a discrete mathematics textbook such as those by Epp (2010), Gersting (2014), or Rosen (2011) for a complete treatment of these topics.

5.1 Propositional Logic

A *proposition* is a meaningful declarative sentence that is either true or false. Propositions are also called *logical statements* or just *statements*. A statement cannot be true at one point in time and false at another time. Here, for example, are two simple propositions, one true and one false:

> Sodium Azide is a poison.
> New York City is the capital of New York state.

Not all statements that can be uttered in a natural language are unambiguously true or false. When a person makes a statement such as, "I like Italian food," there are usually many subtle qualifications to the meaning at play. The speaker might really mean, "I usually like Italian food," or, "I've had Italian food that I liked." The true meaning is either evident from the context of the conversation or can be explored in greater depth by asking clarifying questions. In any case, the speaker almost certainly does not mean he or she definitely likes all Italian food in the world. The original statement is neither completely true nor completely false.

Even mathematical expressions may be ambiguous. We cannot tell whether the expression $x \geq 17$ is true or false as we do not know the value of x. We can

Table 5.1. Connectives of propositional logic

Symbol	Formal name	Informal name
$\neg$	Negation	not
$\wedge$	Conjunction	and
$\vee$	Disjunction	or
$\rightarrow$	Implication	conditional
$\leftrightarrow$	Equivalence	biconditional

turn this expression into a proposition by giving x a value. In Section 5.4, we will show how to use quantifiers to give values to such variables.

Whereas literature, poetry, and humor depend on the emotional impact of ambiguous statements rife with subtle meanings, high-integrity systems must be constructed in more absolute terms. The pilot of an aircraft wants to know that if a certain control is activated, the landing gear will definitely respond. Thus, we are interested in statements with clear truth values.

We use symbols such as $s_1, s_2, s_3, \ldots$ to represent statements. For example, s_1 might represent "The Chicago Cubs won the world series this year," s_2 might represent "The winner of the world series is the best baseball team," and s_3 might represent "The players of the best baseball team have the highest average salary." We can combine these statements using operators called *propositional connectives*. The names and symbols used for these operators are given in Table 5.1 in decreasing order of precedence. As usual, parentheses can be used to clarify or override the normal precedence.

All of the connectives listed in Table 5.1 are infix binary operators except for $\neg$, which is a prefix unary operator. The truth values for these connectives are given in the Table 5.2.

The first two propositional logic connectives in Table 5.2 behave according to their informal descriptions: $\neg s_1$ means "not s_1" and returns true whenever s_1 is false and vice versa, and $s_1 \wedge s_2$ means "s_1 and s_2." It returns true only when both s_1 and s_2 are both true. You are certainly familiar with operators in

Table 5.2. Truth tables for propositional connectives

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \leftrightarrow y$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

various programming languages that carry out negation and conjunction. Here are two assignment statements that illustrate their use in SPARK with Boolean variables A, B, and C:

```
A := not B;
A := B and C;
```

The other connectives require some additional explanation: $\vee$, the or connective, is *inclusive* in the sense that it returns true when either one or both of its operands are true. In common speaking, the word or is sometimes used in an *exclusive* sense ("You can have cookies or candy [but not both]"). In propositional logic this *exclusive or* function can be simulated with a more complex formula:

$$(s_1 \vee s_2) \wedge \neg(s_1 \wedge s_2)$$

Informally, this reads, "You can have (cookies or candy) and not (cookies and candy)." Notice that even though people do not normally speak with parentheses, it is often necessary to add some clarification to the informal expression to avoid ambiguity. Without the parentheses the listener might wonder if the speaker meant, "You can have (cookies or (candy and not cookies)) and candy." In real life, ambiguities of this nature are often resolved using contextual information, which humans are exceptionally good at processing, but misunderstandings do sometimes occur regardless. SPARK provides operators for inclusive and exclusive or:

```
A := B or C;     -- Inclusive or
A := B xor C;    -- Exclusive or
```

The implication connective, $\rightarrow$, captures the meaning of conditional expressions. Let s_1 be the statement, "You work hard in this course," and s_2 be "You will get a good grade." For now we will suppose these statements have well-defined truth values. In that case, the formula $s_1 \rightarrow s_2$ informally translates to "If you work hard in this course, then you will get a good grade." This formula is often read "s_1 implies s_2."

Table 5.2 shows that the only time the conditional expression is false is when the first statement, called the *antecedent*, is true and the second statement, called the *consequent*, is false. In that world, you did work hard and yet you got a poor grade anyway – the conditional expression was not true. In all other cases, however, the conditional expression is considered true. Sometimes this can lead to surprising results. For example, the following statement is true: "If the moon is made of green cheese, then you will win the lottery this year." Because the antecedent is false, it does not matter what the truth value of the consequent

Table 5.3. Logical connectives associated with common English phrases

English phrase	Proposition
A and B, A but B, A also B, A in addition B, A moreover B	$A \wedge B$
A or B	$A \vee B$
if A, then B, A only if B, A implies B, A, therefore B, B follows from A, A is a sufficient condition for B, B is a necessary condition for A	$A \rightarrow B$
A if and only if B, A is a necessary and sufficient condition for B	$A \leftrightarrow B$

might be. In effect, the conditional expression does not apply in that case. Conditional expressions with false antecedents are said to be *vacuously true*.

In SPARK, conditional expressions may be written as if expressions. For example, the value of a determined by the logical statement $b \rightarrow c$ may be written in SPARK as

```
A := (if B then C);
```

Recall from Section 2.1.3 that an if expression without an else clause evaluates to true when the condition is false, just as in the definition of implication. This vacuously true behavior is essential when we prove SPARK verification conditions.

The biconditional connective expresses the idea that the truth value of its operands are always the same. If $s_1 \leftrightarrow s_2$ is true, then s_1 and s_2 have identical truth values in all cases. The phrase "if and only if," as often used in mathematical texts, is translated to the biconditional connective. "P if and only if Q" means $P \leftrightarrow Q$. This property is used to express the concept of *logical equivalence* that we discuss in Section 5.2. SPARK does not have a construct to directly implement the biconditional. Instead, we must make use of a logically equivalent expression described in Section 5.2.

One of the tasks of developing software is the translation of English language expressions into formal expressions. This translation is particularly important when working with logical connectives. Table 5.3 gives a number of common English phrases and their equivalent logical expressions.

It is important to remember that English, like any natural language, is often ambiguous and nuanced. Furthermore, people often speak inaccurately. Use Table 5.3 only as a guide.

Table 5.4. Truth table for $\neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$

Three inputs							
s_1	s_2	s_3	$\neg s_1$	$s_1 \leftrightarrow s_3$	$s_3 \vee (s_1 \leftrightarrow s_3)$	$s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3))$	$\neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$
F	F	F	T	T	T	T	T
F	F	T	T	F	T	T	T
F	T	F	T	T	T	T	T
F	T	T	T	F	T	T	T
T	F	F	F	F	F	T	F
T	F	T	F	T	T	T	F
T	T	F	F	F	F	F	F
T	T	T	F	T	T	T	F

We used Table 5.2 to define the five propositional connectives. Truth tables are also used to show all the possible values of a complex statement. For example, the possible values of the complex statement s_4 defined as

$$s_4 = \neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$$

are given in Table 5.4. This table contains a row for each combination of the three primitive statements (s_1, s_2, s_3) that define s_4. The number of rows in a truth table for a complex statement is 2^n when n is the number of different primitive statements in the definition of the complex statement. The three primitive statements defining s_4 are written as the first three columns of Table 5.4. The eight rows contain all the possible combinations of values of these three inputs. The remaining five columns in the table correspond to the five logical connectives used in the definition of s_4 and are in the order that the connectives would be evaluated from the three inputs.

Certain formulae have the property of being true regardless of the truth values of their constituent statements. These expressions are said to be *tautologies*. A simple example of a tautology is $s_1 \vee \neg s_1$. If s_1 is true, then $s_1 \vee \neg s_1$ is true. And if s_1 is false, $s_1 \vee \neg s_1$ is still true. Although tautologies might at first glance appear to be uninteresting, they play a very important role in validating arguments, as we will see in Section 5.3.

5.2 Logical Equivalence

Look at the fourth column (labeled $\neg s_1$) and the last column of Table 5.4. The values in these two columns are identical. We say that $\neg s_1$ and $\neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$ are *logical equivalents*. The symbol $\Leftrightarrow$ is used to indicate the

Table 5.5. Logical equivalences

Name	*or* form	*and* form
Commutative	$A \vee B \Leftrightarrow B \vee A$	$A \wedge B \Leftrightarrow B \wedge A$
Associative	$(A \vee B) \vee C \Leftrightarrow A \vee (B \vee C)$	$(A \wedge B) \wedge C \Leftrightarrow A \wedge (B \wedge C)$
Distributive	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
De Morgan	$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$	$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
Absorption	$A \vee (A \wedge B) \Leftrightarrow A$	$A \wedge (A \vee B) \Leftrightarrow A$
Idempotent	$A \vee A \Leftrightarrow A$	$A \wedge A \Leftrightarrow A$
Identity	$A \vee \mathit{False} \Leftrightarrow A$	$A \wedge \mathit{True} \Leftrightarrow A$
Universal bound	$A \vee \mathit{True} \Leftrightarrow \mathit{True}$	$A \wedge \mathit{False} \Leftrightarrow \mathit{False}$
Complement	$A \vee \neg A \Leftrightarrow \mathit{True}$	$A \wedge \neg A \Leftrightarrow \mathit{False}$
Double negation		$\neg(\neg A) \Leftrightarrow A$
Conditional as disjunction		$A \rightarrow B \Leftrightarrow \neg A \vee B$
Biconditional as conjunction		$A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$

logical equivalence of two statements as shown here:

$$\neg s_1 \quad \Leftrightarrow \quad \neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$$

We can substitute the value of one statement for the value of the other statement. Another way to look at this particular equivalence is that the statement $\neg s_1 \wedge (s_2 \rightarrow (s_3 \vee (s_1 \leftrightarrow s_3)))$ can be simplified to $\neg s_1$.

Simplification of statements, like the simplification of algebraic expressions, can be accomplished by applying certain rules. The rules we use for simplifying statements are called *logical equivalences*. Table 5.5 summarizes the most commonly used logical equivalences. Each of the logical equivalences in Table 5.5 could be proven by constructing a truth table.

Let us look at some examples to show how these logical equivalences can be used to simplify statements. The name after each line is that of the logical equivalence we used to obtain the line from the previous line. In the first example, we use logical equivalences to reduce the number of logical connectives from four to two. In the second example, we simplify a complex statement to a single primitive statement with no connectives.

$$\begin{aligned} \neg(p \wedge \neg q) \vee q & \\ (\neg p \vee \neg(\neg q)) \vee q & \quad \text{De Morgan} \\ (\neg p \vee q) \vee q & \quad \text{Double negation} \\ \neg p \vee (q \vee q) & \quad \text{Association} \\ \neg p \vee q & \quad \text{Idempotent} \end{aligned}$$

$\neg(q \rightarrow q) \vee (p \wedge q)$	
$\neg(\neg q \vee p) \vee (p \wedge q)$	Conditional as disjunction
$(\neg(\neg q) \wedge \neg p) \vee (p \wedge q)$	De Morgan
$(q \wedge \neg p) \vee (p \wedge q)$	Double negation
$(q \wedge \neg p) \vee (q \wedge p)$	Commutative
$q \wedge (\neg p \vee p)$	Distributive
$q \wedge \mathit{True}$	Compliment
q	Identity

5.3 Arguments and Inference

An argument is an attempt to convince someone of something. In formal logic, an *argument* is expressed by a set of statements called *premises* that together support the truth of another statement called a *conclusion*. Here is an example of an argument with two premises followed by a conclusion:

> If Horace swallowed Sodium Azide, then Horace was poisoned.
> Horace swallowed Sodium Azide.
> Therefore, Horace was poisoned.

All arguments can be expressed in the following form:

$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \ldots \wedge p_n) \rightarrow c$$

where the p's are the premises and c is the conclusion. Informally, this statement says that if all of the premises are true, then the conclusion is true. As premises in a poorly constructed argument can have no relationship to the conclusion, having true premises is not sufficient to show that the conclusion is true. We need to be concerned with the form of the argument. An argument is said to be *valid* if this implication is a tautology. Only when an argument is valid is the conclusion a logical consequence of its premises.

Consider the poisoning argument given at the beginning of this section. If we use s_1 to represent "Horace swallowed Sodium Azide" and s_2 to represent "Horace was poisoned," we can write the argument formally as

$$((s_1 \rightarrow s_2) \wedge s_1) \rightarrow s_2$$

One way to demonstrate that this statement is a tautology, and therefore a valid argument, is to construct a truth table as we have done in Table 5.6. There you can see that no matter what the values of the inputs s_1 and s_2, the value of the argument, $((s_1 \rightarrow s_2) \wedge s_1) \rightarrow s_2$, is true. Note that there is only one row

Table 5.6. Truth table for $((s_1 \to s_2) \wedge s_1) \to s_1$

s_1	s_2	$s_1 \to s_2$	$(s_1 \to s_2) \wedge s_1$	$((s_1 \to s_2) \wedge s_1) \to s_1$
F	F	T	F	T
F	T	T	F	T
T	F	F	F	T
T	T	T	T	T

in this truth table where both of the two premises, $s_1 \to s_2$ and s_1, are true. Because of the nature of implication, when a premise is false, the argument is vacuously true. Thus, to show that an argument is valid, we need only fill in the rows of the truth table in which the premises are all true.

Whereas truth tables provide one method of validating arguments, they are cumbersome when there are large numbers of inputs and premises. Another approach is to construct a formal proof of the validity of an argument. We do this by applying derivation rules called *rules of inference* or *inference rules*. We use these rules to derive new valid statements that follow from previous statements. Inference rules are like "mini-arguments" that are known to be valid. Our poison argument is an example of one of the most common inference rules called modus ponens. It may expressed as follows:

$$\frac{s_1 \to s_2 \qquad s_1}{s_2}$$

Informally this notation means that the conclusion below the line follows from the hypotheses spaced out above the line. More precisely, this notation means that if one forms a conditional expression with the conjunction of the hypotheses as the antecedent and the conclusion as the consequent, that conditional expression is a tautology.

Many useful rules of inference exist. Here are the most commonly used ones:

Modus ponens
$$\frac{A \to B \qquad A}{B}$$

Modus tolens
$$\frac{A \to B \qquad \neg B}{\neg A}$$

Conjunction
$$\frac{A \qquad B}{A \wedge B}$$

Simplification
$$\frac{A \wedge B}{A \qquad B}$$

Addition
$$\frac{A}{A \vee B}$$

Hypothetical syllogism
$$\frac{A \to B \qquad B \to C}{A \to C}$$

Disjunctive syllogism
$$\frac{A \vee B \qquad \neg A}{B}$$

Contrapositive
$$\frac{A \to B}{\neg B \to \neg A}$$

Disjunction elimination
$$\frac{A \vee B \qquad A \to C \qquad B \to C}{C}$$

Resolution
$$\frac{A \vee B \qquad \neg A \vee C}{B \vee C}$$

The simplification rule contains two conclusions. This is a shorthand for two different rules with the same premises and two different conclusions.

Here are two example proofs using rules of inference to demonstrate an argument is valid (the statement defining the argument is a tautology). To accomplish the proof, we start with the premises and use various inference rules and logical equivalences to transform them to the conclusion. The right column of each proof describes how the particular statement on the line was derived from prior lines.

$\neg(s_1 \vee \neg s_2) \wedge (s_2 \rightarrow s_3) \rightarrow (\neg s_1 \wedge s_3)$ – The argument to validate

1.	$\neg(s_1 \vee \neg s_2)$	Premise
2.	$s_2 \rightarrow s_3$	Premise
3.	$\neg s_1 \wedge \neg(\neg s_2)$	De Morgan applied to line 1
4.	$\neg s_1 \wedge s_2$	Double negation applied to line 3
5.	$\neg s_1$	Simplification applied to line 4
6.	s_2	Simplification applied to line 4 (again)
7.	s_3	Modus ponens applied to lines 2 and 6
8.	$\therefore (\neg s_1 \wedge s_3)$	Conjunction applied to lines 5 and 7

$(s_1 \rightarrow s_3) \wedge (s_3 \rightarrow \neg s_2) \wedge s_2 \rightarrow \neg s_1$ – The argument to validate

1.	$s_1 \rightarrow s_3$	Premise
2.	$s_3 \rightarrow \neg s_2$	Premise
3.	s_2	Premise
4.	$s_1 \rightarrow \neg s_2$	Hypothetical syllogism applied to lines 1 and 2
5.	$\neg(\neg s_2)$	Double negative applied to line 3
6.	$\therefore \neg s_1$	Modus tollens applied to lines 4 and 5

This approach to proving argument validity has a direct counterpart in SPARK. GNATprove can generate arguments[1] called *verification conditions* (VCs) for each subprogram. It uses our preconditions as premises and our post-conditions as conclusions. If we can prove these VCs, we have shown that our postconditions always follow from our preconditions. Fortunately, we do not have to manually perform the proofs like we did in the preceding examples. GNATprove makes use of automated theorem provers to accomplish the task.

However, we sometimes need to help out the prover with additional assertions. We will look at the details of program proof in Chapter 6.

What do you think of the following argument?

> If the moon is made of green cheese, then NASA can feed the world.
> The moon is made of green cheese.
> Therefore, NASA can feed the world.

This argument is valid by modus ponens. As we know, however, the second premise is false. That makes the argument statement $(p_1 \wedge p_2) \rightarrow c$ vacuously true. Argument validity is just about argument form or syntax. Most useful arguments have premises that are true. An argument that is valid and has true premises is said to be a *sound argument*. This principle applies to program proof. GNATprove will check that the preconditions we write for our subprograms are always true.

5.4 Predicate Logic

The statements studied so far are fixed statements. However, the interesting logical structure of many useful statements is lost when expressed in such simple terms. For example, a statement such as "all students work hard" might be a legitimate statement but expressing it as such does not convey the significant fact that it is a statement about an entire set of entities.

To express such statements more directly, we need to first parameterize them. Instead of treating a statement as a bare fact s_1, we can give it multiple parameters $s_1(x_1, x_2, \ldots, x_n)$. We call such parameterized statements *predicates*. The values of x_i are taken from a set U called the *universe of discourse* and can be any set that is convenient.

For example, let U be the set of all humans alive on Earth today. Let the predicate $s(x)$ be true if $x \in U$ is a student.[2] Let the predicate $w(x)$ be true if $x \in U$ works hard. Once appropriate arguments are given to a predicate, we will take the resulting statement to be either true or false. For now, we require that all arguments come from the same set U and thus are, in effect, the same type.

If Alice $\in U$, then a statement such as s(Alice) is true if Alice is a student and false otherwise. The statement s(Alice) $\rightarrow$ w(Alice) means informally "if Alice is a student then Alice works hard." This statement is a statement about Alice specifically. It might be true or it might be false. In any case, the truth of a similar statement about Bob $\in U$, s(Bob) $\rightarrow$ w(Bob), might be entirely different.

What can we say about an expression $M = s(x) \rightarrow w(x)$? A *variable* x stands for an as yet unspecified element of U. Without a particular value for x, M is not a statement – it has no truth value. We say that the variable x is *unbound* in M.

To make M into a proper statement, we need to *bind* the variable x. There are two binders that interest us the *universal quantifier*, symbolized with $\forall$, and the *existential quantifier*, symbolized with $\exists$.

The statement $\forall x\,(s(x) \rightarrow w(x))$ reads informally as, "for all x, if x is a student, then x works hard." More precisely, we mean $\forall x \in U\,(s(x) \rightarrow w(x))$, but the universe of discourse is usually evident from context and not normally mentioned explicitly in the formulae. It is important to note that x ranges over the whole set U. Recall that when the antecedent of an implication is false, the implication is vacuously true. So $s(x) \rightarrow w(x)$ is automatically true for any x that is not a student.

SPARK uses the syntax **for all** and the arrow delimiter, $=>$, to implement the universal quantifier. The expression on the right side of the assignment statement

```
A := (for all X in Integer => (if X rem 10 = 0 then X rem 2 = 0));
```

implements the quantified predicate $\forall x \in \mathbb{Z}\,(d(x) \rightarrow e(x))$, where $\mathbb{Z}$ (our universe of discourse) is the set of all integers, $d(x)$ is the predicate "x is evenly divisible by 10," and $e(x)$ is the predicate "x is even." This expression states that any integer that is evenly divided by 10 is even. As you might imagine, true is assigned to variable A.

At the beginning of this section we talked about the statement, "all students work hard," and we now have a formal encoding of that statement using the universal quantifier that exposes the fact that it is a statement about an entire set of entities. We could bury the quantification inside a simple statement by just setting $s_1 = \forall x\,(s(x) \rightarrow w(x))$ and then talk about s_1. There is nothing wrong with this, and at times it is entirely appropriate. However, many interesting proofs will require the more detailed view of the predicate's internal, universally quantified structure.

Now let us look at the use of the existential quantifier to bind statement variables. Consider the informal statement, "some students work hard." This means that there is at least one student who works hard or, equivalently, there exists a student who works hard. This statement can be encoded using the existential quantifier as $s_2 = \exists x\,(s(x) \wedge w(x))$. The truth of s_2 implies that there is at least one entity in U that is both a student and who works hard. The precise number of such entities that satisfy the predicate is not specified; it could even be all of them. Notice that $\exists x\,(s(x) \rightarrow w(x))$ is something entirely different.

To understand it we can use the conditional as a disjunction equivalent (from Table 5.5) on the expression covered by the quantifier to obtain $\exists x\,(\neg s(x) \vee w(x))$. If there is even one entity in U that is not a student, this statement is true. It is clear this has nothing to do with saying that some students work hard.

SPARK uses the syntax **for some** and the arrow delimiter, =>, to implement the existential quantifier. The expression on the right side of the assignment statement (that calls the Boolean function Is_Prime)

```
A := (for some X in Natural => (X rem 2 = 0 and then Is_Prime (X)));
```

implements the quantified predicate $\exists x \in \mathbb{N}\,(e(x) \wedge p(x))$, where $\mathbb{N}$ (our universe of discourse) is the set of all natural numbers, $e(x)$ is the predicate "x is even," and $p(x)$ is the predicate "x is a prime number." This expression states that some natural number is both even and prime. As 2 is a prime number, true is assigned to variable A.

Our examples illustrate a common association between quantifiers and logical connectives. The universal quantifier, $\forall$, is most commonly associated with the implication operator, $\rightarrow$. The existential quantifier, $\exists$, is most commonly associated with the conjunction operator, $\wedge$.

We can simplify quantified expressions by narrowing the domain of the bound variable. For example, our statement, "all students work hard," may be written as either

$$\forall x \in U\,(s(x) \rightarrow w(x)) \quad \text{or} \quad \forall x \in S\,(w(x)),$$

where U is the set of all humans alive on earth today and S is the set of all students (a subset of U). Similarly our statement, "some students work hard," may be written as either

$$\exists x \in U\,(s(x) \wedge w(x)) \quad \text{or} \quad \exists x \in S\,(w(x)).$$

We frequently use types or subtypes to narrow domains when writing quantifiers in SPARK. For example, the subtype declaration

```
subtype Digit is Integer range -9 .. 9;
```

limits the value of the bound variable X in the following quantified expression:

```
A := (for all X in Digit => X ** 2 < 100);
```

How can we determine whether a quantified predicate is true or false? To show that a universally quantified predicate is false, we simply need to find one value of the bound variable that results in the predicate being false. For example, to show that the statement $\forall x\,(s(x) \rightarrow w(x))$ is false, we need to find one student who does not work hard. It takes more effort to show that a universally quantified predicate is true. We need to go through the entire universe

Table 5.7. Quantifiers associated with common English phrases

English phrase	Quantifier
for all, all, every, each, any	use $\forall$
there exists, some, one, at least one	use $\exists$

of discourse to see that there are no values that make our predicate false. In your discrete math class, you probably studied how direct proofs can be used in place of this brute force approach.

With the existential quantifier, we need find only one value that makes the predicate true to say the entire statement is true. To show that it is false, we must go through all of the values in the domain to make sure that none of them satisfy the predicate.

As with logical connectives we need to be able to translate English phrases into logical quantifiers. Our examples should give you some insight into the correspondences. Table 5.7 shows the most common English phrases for our two quantifiers. However, as with Table 5.3, care still needs to be used when translating English statements into formal logical statements.

All of our predicates to this point have had a single parameter. We may have multiple parameter predicates. For example, let the predicate $c(x, y)$ stand for "person x likes cuisine y," where "likes" means that they would be willing to go with a group of friends to a restaurant that specializes in that cuisine. This predicate involves two universal sets of discourse: U, the set of all humans alive today, and F, the set of all possible food cuisines. Table 5.8 gives examples of the use of this two-parameter predicate.

Predicates with multiple parameters are usually used in statements with multiple quantifiers. Table 5.9 provides some examples. Here we use the predicate $c(x, y)$ defined for the previous table and the predicate $l(x, y)$ defined as "person x loves person y." For the second half of the table, the universe of discourse is implied to be U, the set of all people alive today.

Table 5.8. Examples using the predicate $c(x, y)$, "person x likes cuisine y"

Predicate	English translations
c(Bob, Thai)	Bob likes Thai food.
$\forall p \in U\ (c(p, \text{Chinese}))$	Everybody likes Chinese food.
$\forall f \in F\ (c(\text{Mildred}, f))$	Mildred likes all cuisines.
$\exists f \in F\ (\neg c(\text{Horace}, f))$	Horace does not like some cuisine.
$\exists p \in U\ (c(p, \text{Mexican}))$	Somebody likes Mexican food.

Table 5.9. Examples using multiple quantifiers with the predicates $c(x, y)$, "person x likes cuisine y," and $l(x, y)$, "person x loves person y"

Predicate	English translations
$\forall p \in U\ (\forall f \in F\ (c(p, f)))$	Everybody likes all cuisines.
$\forall p \in U\ (\exists f \in F\ (c(p, f)))$	Everybody likes some cuisine.
$\exists p \in U\ (\forall f \in F\ (c(p, f)))$	Somebody likes all cuisines.
$\exists p \in U\ (\exists f \in F\ (c(p, f)))$	Somebody likes some cuisine.
$\forall p\ (l(p, Raymond))$	Everybody loves Raymond.
$\exists p\ (\forall q\ (\neg l(p, q)))$	Somebody loves no one.
$\exists p\ (\forall q\ (\neg l(q, p)))$	There is somebody that no one loves.
$\forall p\ (\exists q\ (l(p, q)))$	Everybody loves somebody.
$\exists p\ (\forall q\ (l(p, q)))$	Somebody loves everybody.
$\forall p\ (\forall q\ (l(p, q) \rightarrow p = q))$	Everybody loves only themselves.

It takes practice to translate quantified statements into English and even more practice to translate English into quantified statements. Each of the three discrete mathematics textbooks referenced at the beginning of this chapter provides a wealth of further examples.

Let us look at some realistic examples of the use of the existential and universal quantifiers in a SPARK program. Here is the specification for a search procedure that determines the location of the first occurrence of a value in an array of values:

```
type Array_Type is array (Positive range <>) of Integer;

procedure Find_First (List     : in  Array_Type;
                      Value    : in  Integer;
                      Position : out Positive)
   with
      Depends => (Position => (List, Value)),
      Pre     => (for some Index in List'Range => List (Index) = Value),
      Post    => (Position in List'Range and then
                    List (Position) = Value and then
                    (for all Index in List'First .. Position - 1 =>
                       List (Index) /= Value));
```

The precondition states that there is at least one occurrence of Value in the array List, $\exists Index\ (List(Index) = Value)$. The predicate variable Index has a domain (universe of discourse) of the range of the array List's index.

The postcondition is a conjunction of three parts. The first part states that the result Position is a legal subscript for the array List. The second part states that Position is the location of Value in List. The third part uses a universal quantifier with a limited domain to state that all the locations before Position do not contain Value.

Here is another search procedure that has a slightly different specification. This time, the value we are looking for may or may not be in the array. Also, the array does not contain any duplicate values. The answers consist of a Boolean telling whether or not the value was found and, if it was found, a position giving its location in the array.

```
procedure Find_Only  (List     : in  Array_Type;
                      Value    : in  Integer;
                      Found    : out Boolean;
                      Position : out Positive)
   with
      Depends => ((Found, Position) => (List, Value)),
      Pre     => (List'Length > 0 and then
                  List'Last < Positive'Last and then
                  (for all J in List'Range =>
                     (for all K in List'Range =>
                        (if List (J) = List (K) then J = K)))),
      Post    => (Position in List'Range and then
                  ((Found and then List (Position) = Value)
                        or else
                   (not Found and then Position = List'Last and then
                   (for all J in List'Range => List (J) /= Value))));
```

We have three preconditions. The first states that there is at least one element in the array. The second states that the upper bound of the array is less than the largest Positive value. These first two preconditions ensure that there is no overflow in any of the arithmetic expressions required in the body. The third precondition states that the array has no duplicate values. This check requires two universal quantifiers. The way to state that there are no duplicates is to compare all pairs of values in the array. If we find an equality in this comparison, the two indexes must be the same. We used the same logic in the last example in Table 5.9. In formal terms this last postcondition may be expressed as

$$\forall j\ (\forall k\ ((Value(j) = Value(k)) \rightarrow (j = k))).$$

The postcondition is more complicated. The first part states that the location returned is a legal subscript of the array. The second part states that either we

found the value in the array at the returned location or we did not find the value in the array. In the case we did not find the value in the array, the location returned is the largest index of the array.

Summary

- A proposition is a meaningful declarative sentence that is either true or false.
- *Logical statement* and *statement* are synonyms of proposition.
- Statements may be represented by symbols such as s_1, s_2, A, and P.
- Statements may be combined using the logical connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), or $\leftrightarrow$ (equivalence).
- In the implication $a \rightarrow b$, a is called the *antecedent*, and b is called the *consequent*.
- An implication with a false antecedent is vacuously true.
- *Truth tables* are used to define logical connectives and to show the possible values of a complex statement.
- A *tautology* is a statement that is always true irregardless of the truth values of any primitive statements defining it.
- Two propositions are *logically equivalent* when they have the same values for all "inputs".
- *Logical equivalences* are used to simplify logical statements.
- A *formal argument* is a set of statements called *premises* that together support the truth of another statement called the *conclusion*.
- Arguments are expressed in the form $(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \ldots \wedge p_n) \rightarrow c$.
- A *valid argument* is one whose implication form is a tautology.
- A truth table can be used to show that an argument is valid.
- *Rules of inference* are valid arguments that may be used to show that other arguments are valid.
- SPARK uses arguments called *verification conditions* to demonstrate that the postconditions of a subprogram may be derived from its preconditions.
- A *sound argument* is a valid argument whose premises are all true.
- A *predicate* is a parameterized statement. We must supply specific values for a predicate's parameters to determine its truthfulness.
- An *unbound variable* is a variable without a value that is used in a predicate. We cannot determine the truthfulness of a predicate with unbound variables.
- We may use the *universal quantifier*, $\forall$, or the *existential quantifier*, $\exists$, to bind a variable in a predicate.

- Converting English language sentences into propositions with quantified predicates and logical connectors is an important skill for software engineers who write SPARK preconditions and postconditions.

Exercises

5.1 Define the following terms.

a. proposition
b. statement
c. propositional connective
d. conjunction
e. disjunction
f. exclusive or
g. implication
h. antecedent
i. consequent
j. vacuously true
k. logical equivalence
l. tautology

5.2 Complete the following truth table to prove the logical equivalence conditional as disjunction, $x \rightarrow y \Leftrightarrow \neg x \vee y$.

x	y	$x \rightarrow y$	$\neg x$	$\neg x \vee y$
F	F			
F	T			
T	F			
T	T			

5.3 Complete the following truth table to prove the absorption logical equivalence, $x \wedge (x \vee y) \Leftrightarrow x$.

x	y	$x \vee y$	$x \wedge (x \vee y)$
F	F		
F	T		
T	F		
T	T		

5.4 Given that p is true, q is false, and r is true, what are the truth values of the following propositions?

a. $\neg(p \wedge q) \vee r$
b. $(p \wedge q) \vee r$
c. $p \wedge (q \vee r)$
d. $\neg p \vee \neg(\neg q \wedge r)$
e. $q \rightarrow (p \vee q)$
f. $(p \vee q) \rightarrow r$
g. $r \rightarrow (\neg p \wedge \neg q)$
h. $q \vee (\neg p \rightarrow r)$

5.5 Translate the statement definition $a = b \leftrightarrow c$ into a SPARK assignment statement using the Boolean variables A, B, and C. You will need to use a logical equivalence from Table 5.5.

5.6 Use the logical equivalences from Table 5.5 to simplify the following propositions. State the name of each equivalence you use as in the examples on page 140.

a. $p \wedge \neg(p \wedge q)$
b. $(p \rightarrow q) \wedge p$
c. $(p \vee q) \wedge (p \vee \neg q)$
d. $p \vee (q \wedge \neg p)$
e. $p \wedge \neg(p \wedge \neg q)$
f. $\neg(p \wedge q) \wedge (p \vee \neg q)$

5.7 Let P be the statement "Roses are red," Q be the statement "Violets are blue," and R be the statement "Sugar is sweet." Translate the following English sentences into formal statements.

a. Roses are red and violets are blue.
b. Roses are red, but sugar is not sweet.
c. It is not true that roses are red and violets are blue.
d. Roses are red, but sugar is sweet.
e. Roses are red only if violets are blue.
f. Roses being red is a sufficient condition for violets being blue.
g. Roses being red is a necessary condition for violets being blue.
h. Roses being red follows from violets being blue.
i. If roses are red and violets are blue, then sugar is sweet.
j. Whenever roses are red, violets are blue and sugar is sweet.
k. Roses are red, and if sugar is sour, then violets are not blue.

5.8 Define the following terms.

a. argument
b. premise
c. conclusion
d. valid argument
e. sound argument
f. inference rule

5.9 Use inference rules and logical equivalences to show that the following arguments are valid. Label each step as we did on page 143.

a. $(A \rightarrow B) \wedge (C \vee \neg B) \wedge A \rightarrow C$
b. $A \wedge (\neg B \rightarrow \neg A) \rightarrow B$
c. $\neg(A \vee \neg B) \wedge (B \rightarrow C) \rightarrow (\neg A \wedge C)$
d. $\neg A \wedge (A \vee B) \rightarrow B$
e. $(\neg A \rightarrow \neg B) \wedge B \wedge (A \rightarrow C) \rightarrow C$

5.10 Given the predicates

$c(x)$ is x is a car,
$m(x)$ is x is a motorcycle, and
$f(x)$ is x is fast,

translate each of the following English sentences into formal statements. The universe of discourse is all things in the world. Recall that $\forall$ is usually

associated with $\rightarrow$ and that $\exists$ is usually associated with $\wedge$. We have given an answer for the first one.

a. All cars are fast. *answer* $\forall x(C(x) \rightarrow F(x))$
b. Some motorcycles are fast.
c. All cars are fast but no motorcycle is fast.
d. Only motorcycles are fast.
e. No car is fast.
f. If every car is fast, then every motorcycle is fast.
g. Some motorcycles are not fast.
h. If no car is fast, then some motorcycles are not fast.

5.11 Given the predicates

$bx)$ is x is a bee,
$f(x)$ is a flower,
$s(x)$ is x stings, and
$p(x, y)$ is x pollinates y.

translate each of the following English sentences into formal statements. The universe of discourse is all things in the world. Recall that $\forall$ is usually associated with $\rightarrow$ and that $\exists$ is usually associated with $\wedge$. We have given an answer for the second one.

a. All bees sting.
b. Some flowers sting. *answer* $\exists x\,(F(x) \wedge S(x))$
c. Only bees sting.
d. Some bees pollinate all flowers.
e. All bees pollinate some flowers.

5.12 The predicate $e(x)$ is "x is an even integer." Use only this predicate and quantifiers to translate the following English sentences into formal statements. The universe of discourse is all integers.

a. Some integers are even.
b. All integers are even.
c. Some integers are odd.
d. Some integers are both even and odd.
e. The sum of an even integer and 12 is an even integer.
f. The sum of any two even integers is an even integer.

5.13 The following function is given an array of percentages and a specific percentage. It returns the number of percentages in the array that are less than the given specific percentage. As you can see from the code, it accomplishes this task by searching for the first value in the array that is greater than or equal to the specific percentage. For this algorithm to work, the values in the array must be in ascending order. Complete the precondition that states this requirement.

```
type Percent       is delta 0.25 range 0.0 .. 100.0;
type Percent_Array is array (Integer range <>) of Percent;

function Num_Below (List  : in Percent_Array;
                    Value : in Percent) return Natural
-- Returns the number of values in List that are less than Value
   with
      Pre => (

is
   Result : Natural;
   Index  : Integer;
begin
   Result := 0;
   Index  := List'First;
   loop
      exit when Index > List'Last or else List (Index) >= Value;
      Result := Result + 1;
      Index  := Index + 1;
   end loop;
   return Result;
end Num_Below;
```

5.14 Add a second precondition to the function in Exercise 5.13 that states that List contains at least one value.