Chapter 3

Discrete Math for Testers

More than any other life cycle activity, testing lends itself to mathematical description and analysis. In this chapter and in the next, testers will find the mathematics they need. Following the craftsperson metaphor, the mathematical topics presented here are tools; a testing craftsperson should know how to use them well. With these tools, a tester gains rigor, precision, and efficiency—all of which improve testing. The "for testers" part of the chapter title is important: this chapter is written for testers who either have a sketchy math background or have forgotten some of the basics. Serious mathematicians (or maybe just those who take themselves seriously) will likely be annoyed by the informal discussion here. If you are already comfortable with the topics in this chapter, skip to the next chapter and start right in on graph theory.

In general, discrete mathematics is more applicable to functional testing, while graph theory pertains more to structural testing. "Discrete" raises a question: What might be indiscrete about mathematics? The mathematical antonym is continuous, as in calculus, which software developers (and testers) seldom use. Discrete math includes set theory, functions, relations, propositional logic, and probability theory, each of which is discussed here.

3.1 Set Theory

How embarrassing to admit, after all the lofty expiation of rigor and precision, that no explicit definition of a set exists. This is really a nuisance because set theory is central to these two chapters on math. At this point, mathematicians make an important distinction: naive versus axiomatic set theory. In naive set theory, a set is recognized as a primitive term, much like point and line are primitive concepts in geometry. Here are some synonyms for *set*: *collection*, *group*, *bunch* — you get the idea. The important thing about a set is that it lets us refer to several things as a group, or a whole. For example, we might wish to refer to the set of months that have exactly 30 days (we need this set when we test the NextDate function from Chapter 2). In set theory notation, we write:

$$M1 = \{\text{April, June, September, November}\}$$

and we read this notation as "M1 is the set whose elements are the months April, June, September, November."

3.1.1 Set Membership

The items in a set are called elements or members of the set, and this relationship is denoted by the symbol ∈. Thus, we could write April ∈ M1. When something is not a member of a set, we use the symbol ∉, so we might write December ∉ M1.

3.1.2 Set Definition

A set is defined in three ways: by simply listing its elements, by giving a decision rule, or by constructing a set from other sets. The listing option works well for sets with only a few elements as well as for sets in which the elements obey an obvious pattern. We used this method in defining M1 above. We might define the set of allowable years in the NextDate program as follows:

$$Y = \{1812, 1813, 1814, \ldots, 2011, 2012\}$$

When we define a set by listing its elements, the order of the elements is irrelevant. We will see why when we discuss set equality. The decision rule approach is more complicated, and this complexity carries both advantages and penalties. We could define the years for NextDate as

$$Y = \{year : 1812 \leq year \leq 2012\}$$

which reads "Y is the set of all years such that [the colon is "such that"] the years are between 1812 and 2012 inclusive." When a decision rule is used to define a set, the rule must be unambiguous. Given any possible value of year, we can therefore determine whether that year is in our set Y.

The advantage of defining sets with decision rules is that the unambiguity requirement forces clarity. Experienced testers have encountered "untestable requirements." Many times, the reason that such requirements cannot be tested boils down to an ambiguous decision rule. In our triangle program, for example, suppose we defined a set as follows:

$$N = \{t : t \text{ is a nearly equilateral triangle}\}$$

We might say that the triangle with sides (500, 500, 501) is an element of N, but how would we treat the triangles with sides (50, 50, 51) or (5, 5, 6)?

A second advantage of defining sets with decision rules is that we might be interested in sets where the elements are difficult to list. In the commission problem, for example, we might be interested in the following set:

$$S = \{sales : \text{the 15\% commission rate applies to the total sale}\}$$

We cannot easily write down the elements of this set, but given a particular value for sale, we can easily apply the decision rule.

The main disadvantage of decision rules is that they can become logically complex, particularly when they are expressed with the predicate calculus quantifiers ∃ ("there exists") and ∀ ("for all"). If everyone understands this notation, the precision is helpful. Too often customers are overwhelmed by statements with these quantifiers. A second problem with decision rules has to do with self-reference. This is interesting, but it really has very little application for testers.

The problem arises when a decision rule refers to itself, which is a circularity. As an example, the Barber of Seville "is the man who shaves everyone who does not shave himself."

3.1.3 The Empty Set

The empty set, denoted by the symbol ∅, occupies a special place in set theory. The empty set contains no elements. At this point, mathematicians will digress to prove a lot of facts about empty sets:

- The empty set is unique; that is, there cannot be two empty sets (we will take their word for it).
- ∅, {∅}, {{∅}}, are all different sets (we will not need this).

It is useful to note that when a set is defined by a decision rule that is always false, the set is empty. For instance,

$$\emptyset = \{\text{year} : 2012 \leq \text{year} \leq 1812\}$$

3.1.4 Venn Diagrams

Sets are commonly pictured by Venn diagrams, as in Chapter 1, when we discussed sets of specified and programmed behaviors. In a Venn diagram, a set is depicted as a circle; points in the interior of the circle correspond to elements of the set. Then, we might draw our set M1 of 30-day months as in Figure 3.1.

Venn diagrams communicate various set relationships in an intuitive way, but some picky questions arise. What about finite versus infinite sets? Both can be drawn as Venn diagrams; in the case of finite sets, we cannot assume that every interior point corresponds to a set element. We do not need to worry about this, but it is helpful to know the limitations. Sometimes, we will find it helpful to label specific elements.

Another sticking point has to do with the empty set. How do we show that a set, or maybe a portion of a set, is empty? The common answer is to shade empty regions, but this is often contradicted by other uses in which shading is used to highlight regions of interest. The best practice is to provide a legend that clarifies the intended meaning of shaded areas.

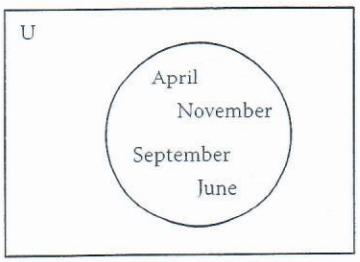

Figure 3.1 Venn diagram of the set of 30-day months.

It is often helpful to think of all the sets in a discussion as subsets of some larger set, known as the universe of discourse. We did this in Chapter 1 when we chose the set of all program behaviors as our universe of discourse. The universe of discourse can usually be guessed from given sets. In Figure 3.1, most people would take the universe of discourse to be the set of all months in a year. Testers should be aware that assumed universes of discourse are often sources of confusion. As such, they constitute a subtle point of miscommunication between customers and developers.

3.1.5 Set Operations

Much of the expressive power of set theory comes from basic operations on sets: union, intersection, and complement. Other handy operations are used: relative complement, symmetric difference, and Cartesian product. Each of these is defined next. In each of these definitions, we begin with two sets, A and B, contained in some universe of discourse U. The definitions use logical connectives from propositional calculus: and ($\wedge$), or ($\vee$), exclusive-or ($\oplus$), and not ($\neg$).

Definition
Given sets A and B,

Their *union* is the set $A \cup B = \{x : x \in A \vee x \in B\}$
Their *intersection* is the set $A \cap B = \{x : x \in A \wedge x \in B\}$
The *complement* of A is the set $A' = \{x : x \notin A\}$
The *relative complement of B with respect to A* is the set $A - B = \{x : x \in A \wedge x \notin B\}$
The *symmetric difference of A and B* is the set $A \oplus B = \{x : x \in A \oplus x \in B\}$

Venn diagrams for these sets are shown in Figure 3.2.

The intuitive expressive power of Venn diagrams is very useful for describing relationships among test cases and among items to be tested. Looking at the Venn diagrams in Figure 3.2, we might guess that

$$A \oplus B = (A \cup B) - (A \cap B)$$

This is the case, and we could prove it with propositional logic.

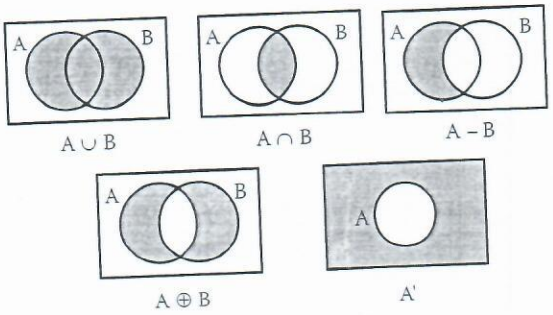

Figure 3.2 Venn diagrams of basic sets.

Venn diagrams are used elsewhere in software development: together with directed graphs, they are the basis of the StateCharts notation, which is among the most rigorous specification techniques supported by CASE technology. StateCharts are also the control notation chosen for the Unified Modeling Language (UML) from IBM Corp. and the Object Management Group.

The Cartesian product (also known as the cross-product) of two sets is more complex; it depends on the notion of ordered pairs, which are two element sets in which the order of the elements is important. The usual notation for unordered and ordered pairs is:

Unordered pair:	(a, b)
Ordered pair:	< a, b >

The difference is that, for $a \neq b$,

$$(a, b) = (b, a)$$

but

$$< a, b > \neq < b, a >$$

This distinction is important to the material in Chapter 4; as we shall see, the fundamental difference between ordinary and directed graphs is exactly the difference between unordered and ordered pairs.

Definition
The *Cartesian product of two sets A and B* is the set

$$A \times B = \{<x, y> : x \in A \wedge y \in B\}$$

Venn diagrams do not show Cartesian products, so we will look at a short example. The Cartesian product of the sets A = {1, 2, 3} and B = {w, x, y, z} is the set

$$A \times B = \{<1, w>, <1, x>, <1, y>, <1, z>, <2, w>, <2, x>, <2, y>, <2, z>, <3, w>, <3, x>, <3, y>, <3, z>\}$$

The Cartesian product has an intuitive connection with arithmetic. The cardinality of a set A is the number of elements in A and is denoted by |A|. (Some authors prefer Card(A).) For sets A and B, $|A \times B| = |A| \times |B|$. When we study functional testing in Chapter 5, we will use the Cartesian product to describe test cases for programs with several input variables. The multiplicative property of the Cartesian product means that this form of testing generates a very large number of test cases.

3.1.6 Set Relations

We use set operations to construct interesting new sets from existing sets. When we do, we often would like to know something about the way the new and the old sets are related. Given two sets, A and B, we define three fundamental set relationships:

Definition

A is a *subset* of B, written $A \subseteq B$, if and only if (iff) $a \in A \Rightarrow a \in B$
A is a *proper subset* of B, written $A \subset B$, iff $A \subseteq B \wedge B - A \neq \varnothing$
A and B are *equal sets*, written $A = B$, iff $A \subseteq B \wedge B \subseteq A$

In plain English, set A is a subset of set B if every element of A is also an element of B. To be a proper subset of B, A must be a subset of B and there must be some element in B that is not an element of A. Finally, the sets A and B are equal if each is a subset of the other.

3.1.7 Set Partitions

A partition of a set is a very special situation that is extremely important for testers. Partitions have several analogs in everyday life: we might put up partitions to separate an office area into individual offices; we also encounter political partitions when a state is divided up into legislative districts. In both of these, notice that the sense of "partition" is to divide up a whole into pieces such that everything is in some piece and nothing is left out. More formally:

Definition
Given a set A, and a set of subsets $A_1, A_2, \ldots, A_n$ of A, the subsets are a *partition of A* iff

$$A_1 \cup A_2 \cup \ldots \cup A_n = A$$

and

$$i \neq j \Rightarrow A_i \cap A_j = \varnothing$$

Because a partition is a set of subsets, we frequently refer to individual subsets as elements of the partition.

The two parts of this definition are important for testers. The first part guarantees that every element of B is in some subset, while the second part guarantees that no element of B is in two of the subsets.

This corresponds well with the legislative districts example: everyone is represented by some legislator, and nobody is represented by two legislators. A jigsaw puzzle is another good example of a partition; in fact, Venn diagrams of partitions are often drawn like puzzles, as in Figure 3.3.

Partitions are helpful to testers because the two definitional properties yield important assurances: completeness (everything is somewhere) and nonredundancy. When we study functional

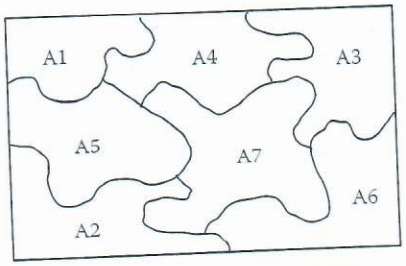

Figure 3.3 Venn diagram of a partition.

testing, we shall see that its inherent weakness is the vulnerability to both gaps and redundancies: some things may remain untested, while others are tested repeatedly. One of the difficulties of functional testing centers on finding an appropriate partition. In the triangle program, for example, the universe of discourse is the set of all triplets of positive integers. (Note that this is actually a Cartesian product of the set of positive integers with itself three times.) We might partition this universe three ways:

1. Into triangles and nontriangles
2. Into equilateral, isosceles, scalene, and nontriangles
3. Into equilateral, isosceles, scalene, right, and nontriangles

At first these partitions seem okay, but there is a problem with the last partition. The sets of scalene and right triangles are not disjoint (the triangle with sides 3, 4, 5 is a right triangle that is scalene).

3.1.8 Set Identities

Set operations and relations, when taken together, yield an important class of set identities that can be used to algebraically simplify complex set expressions. Math students usually have to derive all these; we will just list them and (occasionally) use them.

Name	*Expression*
Identity laws	$A \cup \varnothing = A$
	$A \cap U = A$
Domination laws	$A \cup U = U$
	$A \cap \varnothing = \varnothing$
Idempotent laws	$A \cup A = A$
	$A \cap A = A$
Complementation laws	$(A')' = A$
Commutative laws	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
Associative laws	$A \cup (B \cup C) = (A \cup B) \cup C$
	$A \cap (B \cap C) = (A \cap B) \cap C$
Distributive laws	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
DeMorgan's laws	$(A \cup B)' = A' \cap B'$
	$(A \cap B)' = A' \cup B'$

3.2 Functions

Functions are a central notion to software development and testing. The whole functional decomposition paradigm, for example, implicitly uses the mathematical notion of a function. We make

Informally, a function associates elements of sets. In the NextDate program, for example, the function of a given date is the date of the following day, and in the triangle problem, the function of three input integers is the kind of triangle formed by sides with those lengths. In the commission problem, the salesperson's commission is a function of sales, which in turn is a function of the number of locks, stocks, and barrels sold. Functions in the ATM system are much more complex; not surprisingly, this will add complexity to the testing.

Any program can be thought of as a function that associates its outputs with its inputs. In the mathematical formulation of a function, the inputs are the domain and the outputs are the range of the function.

Definition

Given sets A and B, a *function* f is a subset of $A \times B$ such that, for $a_i, a_j \in A$, $b_i, b_j \in B$, and $f(a_i) = b_i$, $f(a_j) = b_j$, $b_i \neq b_j \Rightarrow a_i \neq a_j$.

Formal definitions like this one are notoriously terse, so let us take a closer look. The inputs to the function f are elements of the set A, and the outputs of f are elements of B. What the definition says is that the function f is "well behaved" in the sense that an element in A is never associated with more than one element of B. (If this could happen, how would we ever test such a function? This is an example of nondeterminism.)

3.2.1 Domain and Range

In the definition just given, the set A is the domain of the function f, and the set B is the range. Because input and output have a "natural" order, it is an easy step to say that a function f is really a set of ordered pairs in which the first element is from the domain and the second element is from the range. Here are two common notations for function:

$$f : A \rightarrow B$$

$$f \subseteq A \times B$$

We have not put any restrictions on the sets A and B in this definition. We could have A = B, and either A or B could be a Cartesian product of other sets.

3.2.2 Function Types

Functions are further described by particulars of the mapping. In the definition below, we start with a function $f : A \rightarrow B$, and we define the set:

$$f(A) = \{b_i \in B : b_i = f(a_i) \text{ for some } a_i \in A\}$$

This set is sometimes called the image of A under f.

Definition

f is a *function from A onto B* iff f(A) = B.
f is a *function from A into B* iff $f(A) \subset B$ (note the proper subset here).
f is a *one-to-one function from A to B* iff, for all $a_i, a_j \in A$, $a_i \neq a_j \Rightarrow f(a_i) \neq f(a_j)$.
f is a *many-to-one function from A to B* iff there exists $a_i, a_j \in A$, $a_i \neq a_j$ such that $f(a_i) = f(a_j)$.

Back to plain English, if f is a function from A onto B, we know that every element of B is associated with some element of A. If f is a function from A into B, we know that there is at least one element of B that is not associated with an element of A. One-to-one functions guarantee a form of uniqueness: distinct domain elements are never mapped to the same range element. (Notice this is the inverse of the well-behaved attribute described earlier.) If a function is not one-to-one, it is many-to-one; that is, more than one domain element can be mapped to the same range element. In these terms, the well-behaved requirement prohibits functions from being one-to-many. Testers familiar with relational databases will recognize that all these possibilities (one-to-one, one-to-many, many-to-one, and many-to-many) are allowed for relations.

Referring again to our testing examples, suppose we take A, B, and C to be sets of dates for the NextDate program, where:

$$A = \{\text{date} : 1 \text{ January } 1812 \leq \text{date} \leq 31 \text{ December } 2012\}$$

$$B = \{\text{date} : 2 \text{ January } 1812 \leq \text{date} \leq 1 \text{ January } 2013\}$$

$$C = A \cup B$$

Now, NextDate : $A \rightarrow B$ is a one-to-one onto function, and NextDate : $A \rightarrow C$ is a one-to-one into function.

It makes no sense for NextDate to be many-to-one, but it is easy to see how the triangle problem can be many-to-one. When a function is one-to-one and onto, such as NextDate : $A \rightarrow B$ previously, each element of the domain corresponds to exactly one element of the range; conversely, each element of the range corresponds to exactly one element of the domain. When this happens, it is always possible to find an inverse function (see the YesterDate exercise in Chapter 2) that is one-to-one from the range back to the domain.

All this is important for testing. The into versus onto distinction has implications for domain- and range-based functional testing, and one-to-one functions may require much more testing than many-to-one functions.

3.2.3 Function Composition

Suppose we have sets and functions such that the range of one is the domain of the next:

$$f : A \rightarrow B$$

$$g : B \rightarrow C$$

$$h : C \rightarrow D$$

When this occurs, we can compose the functions. To do this, let us refer to specific elements of the domain and range sets $a \in A$, $b \in B$, $c \in C$, $d \in D$, and suppose that $f(a) = b$, $g(b) = c$, and $h(c) = d$. Now the composition of functions h, g, and f is

$$\begin{aligned} h \circ g \circ f(a) &= h(g(f(a))) \\ &= h(g(b)) \\ &= h(c) \end{aligned}$$

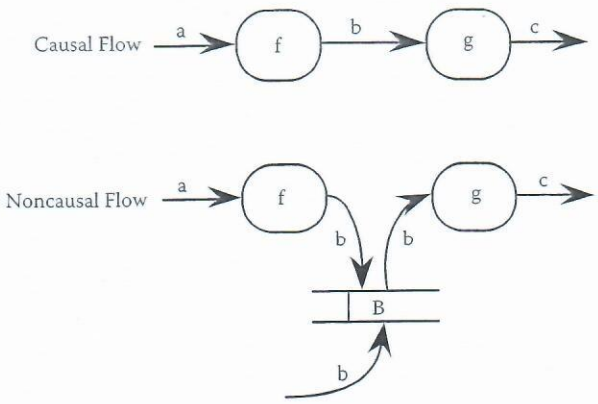

Figure 3.4 Causal and noncausal flows in a dataflow diagram.

Function composition is a very common practice in software development; it is inherent in the process of defining procedures and subroutines. We have an example of it in the commission program, in which

$$f_1(\text{locks, stocks, barrels}) = \text{sales}$$

$$f_2(\text{sales}) = \text{commission}$$

so

$$f_2(f_1(\text{locks, stocks, barrels}) = \text{commission}$$

Composed chains of functions can be problematic for testers, particularly when the range of one function is a proper subset of the domain of the next function in the chain. Figure 3.4 shows how this can happen in a program defined by a dataflow diagram.

In the causal flow, the composition $g \circ f(a)$ (which we know to be g(b), which yields c) is a rather assembly line-like process. In the noncausal flow, the possibility of more than one source of b values for the data store B raises two problems for testers. Multiple sources of b values might raise problems of domain/range compatibility, and even if this is not a problem, there might be timing anomalies with respect to b values. (What if g used an "old" b value?)

A special case of composition can be used, which helps testers in a curious way. Recall we discussed how one-to-one onto functions always have an inverse function. It turns out that this inverse function is unique and is guaranteed to exist (again, the math folks would prove this). If f is a one-to-one function from A onto B, we denote its unique inverse by f^{-1}. It turns out that for $a \in A$ and $b \in B$, $f^{-1} \circ f(a) = a$ and $f \circ f^{-1}(b) = b$. The NextDate and YesterDate programs are such inverses. The way this helps testers is that, for a given function, its inverse acts as a cross-check, and this can often expedite the identification of functional test cases.

3.3 Relations

Functions are a special case of a relation: both are subsets of some Cartesian product, but in the case of functions, we have the well-behaved requirement that says that a domain element cannot be associated with more than one range element. This is borne out in everyday usage: when we say

something "is a function" of something else, our intent is that there is a deterministic relationship present. Not all relationships are strictly functional. Consider the mapping between a set of patients and a set of physicians. One patient may be treated by several physicians, and one physician may treat several patients — a many-to-many mapping.

3.3.1 Relations among Sets

Definition

Given two sets A and B, a *relation R* is a subset of the Cartesian product $A \times B$.

Two notations are popular; when we wish to speak about the entire relation, we usually just write $R \subseteq A \times B$; for specific elements $a_i \in A$, $b_i \in B$, we write $a_i \, R \, b_i$. Most math texts omit treatment of relations; we are interested in them because they are essential to both data modeling and object-oriented analysis.

Next, we have to explain an overloaded term — cardinality. Recall that, as it applies to sets, cardinality refers to the number of elements in a set. Because a relation is also a set, we might expect that the cardinality of a relation refers to how many ordered pairs are in the set $R \subseteq A \times B$. Unfortunately, this is not the case.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the *cardinality of relation R* is:

One-to-one iff R is a one-to-one function from A to B
Many-to-one iff R is a many-to-one function from A to B
One-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is, $\langle a, b_i \rangle \in R$ and $\langle a, b_j \rangle \in R$
Many-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is, $\langle a, b_i \rangle \in R$ and $\langle a, b_j \rangle \in R$, and at least one element $b \in B$ is in two ordered pairs in R, that is, $\langle a_i, b \rangle \in R$ and $\langle a_j, b \rangle \in R$

The distinction between functions into and onto their range has an analog in relations — the notion of participation.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the *participation of relation R* is:

Total iff every element of A is in some ordered pair in R
Partial iff some element of A is not in some ordered pair in R
Onto iff every element of B is in some ordered pair in R
Into iff some element of B is not in some ordered pair in R

In plain English, a relation is total if it applies to every element of A and partial if it does not apply to every element. Another term for this distinction is mandatory versus optional participation. Similarly, a relation is onto if it applies to every element of B and into if it does not. The parallelism between total/partial and onto/into is curious and deserves special mention here. From the standpoint of relational database theory, no reason exists for this; in fact, a compelling reason

exists to avoid this distinction. Data modeling is essentially declarative, while process modeling is essentially imperative. The parallel sets of terms force a direction on relations, when in fact no need exists for the directionality. Part of this is a likely holdover from the fact that Cartesian products consist of ordered pairs, which clearly have a first and second element.

So far, we have only considered relations between two sets. Extending relations to three or more sets is more complicated than simply the Cartesian product. Suppose, for example, we had three sets, A, B, and C, and a relation $R \subseteq A \times B \times C$. Do we intend the relation to be strictly among three elements, or is it between one element and an ordered pair (there would be three possibilities here)? This line of thinking also needs to be applied to the definitions of cardinality and participation. It is straightforward for participation, but cardinality is essentially a binary property. (Suppose, for example, the relation is one-to-one from A to B and is many-to-one from A to C.) We discussed a three-way relation in Chapter 1, when we examined the relationships among specified, implemented, and tested program behaviors. We would like to have some form of totality between test cases and specification–implementation pairs; we will revisit this when we study functional and structural testing.

Testers need to be concerned with the definitions of relations because they bear directly on software properties to be tested. The onto/into distinction, for example, bears directly on what we will call output-based functional testing. The mandatory–optional distinction is the essence of exception handling, which also has implications for testers.

3.3.2 Relations on a Single Set

Two important mathematical relations are used, both of which are defined on a single set: ordering relations and equivalence relations. Both are defined with respect to specific properties of relations.

Let A be a set, and let $R \subseteq A \times A$ be a relation defined on A, with $\langle a, a\rangle$, $\langle a, b\rangle$, $\langle b, a\rangle$, $\langle b, c\rangle$, $\langle a, c\rangle \in R$. Relations have four special attributes:

Definition
A relation $R \subseteq A \times A$ is:

Reflexive iff for all $a \in A$, $\langle a, a\rangle \in R$
Symmetric iff $\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \in R$
Antisymmetric iff $\langle a, b\rangle, \langle b, a\rangle \in R \Rightarrow a = b$
Transitive iff $\langle a, b\rangle, \langle b, c\rangle \in R \Rightarrow \langle a, c\rangle \in R$

Family relationships are nice examples of these properties. You might want to think about the following relationships and decide for yourself which attributes apply: brother of, sibling of, and ancestor of. Now we can define the two important relations.

Definition
A relation $R \subseteq A \times A$ is an *ordering relation* if R is reflexive, antisymmetric, and transitive.

Ordering relations have a sense of direction; some common ordering relations are older than, $\geq$, $\Rightarrow$, and ancestor of. (The reflexive part usually requires some fudging — we really should say not younger than and not a descendant of.) Ordering relations are a common occurrence in software: data access techniques, hashing codes, tree structures, and arrays are all situations in which ordering relations are used.

The power set of a given set is the set of all subsets of the given set. The power set of the set A is denoted P(A). The subset relation $\subseteq$ is an ordering relation on P(A), because it is reflexive (any set is trivially a subset of itself), antisymmetric (the definition of set equality), and transitive.

Definition

A relation $R \subseteq A \times A$ is an *equivalence relation* if R is reflexive, symmetric, and transitive.

Mathematics is full of equivalence relations: equality and congruence are two quick examples. A very important connection exists between equivalence relations and partitions of a set. Suppose we have some partition $A_1, A_2, \ldots, A_n$ of a set B, and we say that two elements, b_1 and b_2 of B, are related (i.e., $b_1 R b_2$) if b_1 and b_2 are in the same partition element. This relation is reflexive (any element is in its own partition), symmetric (if b_1 and b_2 are in a partition element, then b_2 and b_1 are), and transitive (if b_1 and b_2 are in the same set, and if b_2 and b_3 are in the same set, then b_1 and b_3 are in the same set). The relation defined from the partition is called the equivalence relation induced by the partition. The converse process works in the same way. If we start with an equivalence relation defined on a set, we can define subsets according to elements that are related to each other. This turns out to be a partition and is called the partition induced by the equivalence relation. The sets in this partition are known as equivalence classes. The end result is that partitions and equivalence relations are interchangeable, and this becomes a powerful concept for testers. Recall that the two properties of a partition are notions of completeness and nonredundancy. When translated into testing situations, these notions allow testers to make powerful, absolute statements about the extent to which a software item has been tested. In addition, great efficiency follows from testing just one element of an equivalence class and assuming that the remaining elements will behave similarly.

3.4 Propositional Logic

We have already been using propositional logic notation; if you were perplexed by this usage definition before, you are not alone. Set theory and propositional logic have a chicken-and-egg relationship — it is hard to decide which should be discussed first. Just as sets are taken as primitive terms and are therefore not defined, we take propositions to be primitive terms. A proposition is a sentence that is either true or false, and we call these the truth values of the proposition. Furthermore, propositions are unambiguous: given a proposition, it is always possible to tell whether it is true or false. The sentence "Mathematics is difficult" would not qualify as a proposition because of the ambiguity. There are also temporal and spatial aspects of propositions. For example, "It is raining" may be true at some times and false at others. In addition, it may be true for one person and false for another at the same time but different locations.

We usually denote propositions with lowercase letters p, q, and r. Propositional logic has operations, expressions, and identities that are very similar (in fact, they are isomorphic) to set theory.

3.4.1 Logical Operators

Logical operators (also known as logical connectives or operations) are defined in terms of their effect on the truth values of the propositions to which they are applied. This is easy; only two values are used: T (for true) and F (for false). Arithmetic operators could also be defined this way (in fact, that is how they are taught to children), but the tables become too large. The three basic logical operators are and ($\wedge$), or ($\vee$), and not ($\neg$); these are sometimes called conjunction, disjunction, and negation. Negation is the only unary (one operand) logical operator; the others are all binary.

p	q	$p \wedge q$	$p \vee q$	$\neg p$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Conjunction and disjunction are familiar in everyday life: a conjunction is true only when all components are true, and a disjunction is true if at least one component is true. Negations also behave as we expect. Two other common connectives are used: exclusive-or ($\oplus$) and IF-THEN ($\rightarrow$). They are defined as follows:

p	q	$p \oplus q$	$p \rightarrow q$
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T

An exclusive-or is true only when one of the propositions is true, while a disjunction (or inclusive-or) is true when both propositions are true. The IF-THEN connective usually causes the most difficulty. The easy view is that this is just a definition, but because the other connectives all transfer nicely to natural language, we have similar expectations for IF-THEN. The quick answer is that the IF-THEN connective is closely related to the process of deduction: in a valid deductive syllogism, we can say "if premises, then conclusion" and the IF-THEN statement will be a tautology.

3.4.2 Logical Expressions

We use logical operators to build logical expressions in exactly the same way that we use arithmetic operators to build algebraic expressions. We can specify the order in which operators are applied with the usual conventions on parentheses, or we can employ a precedence order (negation first, then conjunction followed by disjunction). Given a logical expression, we can always find its truth table by "building up" to it following the order determined by the parentheses. For example, the expression $\neg ((p \rightarrow q) \wedge (q \rightarrow p))$ has the following truth table:

p	q	$p \rightarrow q$	$q \rightarrow p$	$(p \rightarrow q) \wedge (q \rightarrow p)$	$\neg((p \rightarrow q) \wedge (q \rightarrow p))$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	T	F

3.4.3 Logical Equivalence

The notions of arithmetic equality and identical sets have analogs in propositional logic. Notice that the expressions $\neg((p \rightarrow q) \wedge (q \rightarrow p))$ and $p \oplus q$ have identical truth tables. This means that no matter what truth values are given to the base propositions p and q, these expressions will always have the same truth value. This property can be defined in several ways; we use the simplest.

Definition

Two propositions *p and q are logically equivalent* (denoted $p \Leftrightarrow q$) iff their truth tables are identical.

By the way, the curious "iff" abbreviation we have been using for "if and only if" is sometimes called the biconditional, so the proposition p iff q is really $(p \rightarrow q) \wedge (q \rightarrow p)$, which is denoted $p \leftrightarrow q$.

Definition

A proposition that is always true is a *tautology*; a proposition that is always false is a *contradiction*.

To be a tautology or a contradiction, a proposition must contain at least one connective and two or more primitive propositions. We sometimes denote a tautology as a proposition T and a contradiction as a proposition F. We can now state several laws that are direct analogs of the ones we had for sets.

Law	*Expression*
Identity	$p \wedge T \Leftrightarrow p$
	$p \vee F \Leftrightarrow p$
Domination	$p \vee T \Leftrightarrow T$
	$p \wedge F \Leftrightarrow F$
Idempotent	$p \wedge p \Leftrightarrow p$
	$p \vee p \Leftrightarrow p$
Complementation	$\neg(\neg p) \Leftrightarrow p$
Commutative	$p \wedge q \Leftrightarrow q \wedge p$
	$p \vee q \Leftrightarrow q \vee p$
Associative	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
Distributive	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
DeMorgan's	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
	$\neg(p \vee q) \Leftrightarrow \neg p \subseteq \neg q$

3.5 Probability Theory

We will have two occasions to use probability theory in our study of software testing: one deals with the probability that a particular path of statements executes, and the other generalizes this to a popular industrial concept called an operational profile (see Chapter 14). Because of this limited use, we will only cover the rudiments here.