The Art of Constraint Programming

Todd Ebert

Contents

Chapter 1

Mathematical Preliminaries

Writing a constraint program for a problem means defining variables on which the problem depends, and providing a set of constraint statements that the assigned variables must satisfy. Then finding a solution is reduced to the problem of finding a variable assignment that satisfies each constraint statement. Thus, constraint programming represents a methodology for modeling problems. By *modeling* we mean representing a problem in an abstract mathematical/logical form that lends itself to automated analysis. In this chapter we review the mathematical structures that are needed to create constraint-programming models, including constants, variables, propositional logic, sets, functions, predicate logic, and expressions. We summarize the importance of each of these ideas in relation to constraint programming.

Expression mathematical/logical formula to which a constraint gets translated

Function the fundamental building block for expressions

Constant a type of function whose output never changes

Variable a type of function that allows for an expression to assume a range of different values

Set defines the domain of a variable and, more generally, the domain of a function

Propositional Logic used for creating logical expressions that depend only on Boolean variables

Predicate Logic used for creating logical expressions that may depend on variables of any type

The reader who is already familiar with these ideas may want to skim this chapter to ensure familiarity with all the definitions.

1.1 Sets

A **set** represents a collection of items, where each item is called a **member** of the set.

The most common way to represent a set is by using **list notation**, where the set members are listed one-by-one, and the list is delimited by braces. For example,

$$\{2, 3, 5, 7, 11\}$$

uses list notation to describe the set consisting of all prime numbers that do not exceed 11. Note that the order in which the members are listed does not matter. Indeed the sets $\{2,3,5,7,11\}$ and $\{3,11,5,2,7\}$ are identical. Also, each member occurs only *once* in the set, meaning that, e.g. $\{1,2,2,3,3,3\} = \{1,2,3\}$.

The set having no members is called the **empty set**, and is denoted by $\emptyset$.

It is often convenient to provide a name or label for a given set. For example, $\mathcal{I}$ is used to denote the set whose members are the numbers $0, \pm 1, \pm 2, \ldots$. This book uses the convention that set names begin with a capital letter. Moreover, capital letters $A, B, C, \ldots$ are used when denoting an *abstract set*, meaning some arbitrary set whose members have not been specified. In later sections and chapters we shall see several other methods for naming and representing a set.

We may use the **membership symbol** $\in$ to indicate that an item is a member of some set. For example, $-2 \in \mathcal{I}$ asserts that -2 is a member of $\mathcal{I}$. On the other hand, $3.14 \notin \mathcal{I}$ asserts that 3.14 is *not* a member of $\mathcal{I}$.

Sometimes we need to use **informal list notation** by including the **ellipsis** $\ldots$ to indicate that a pattern is to be continued up to some value. The beginning pattern usually consists of one or more members, followed by an ellipsis, and ending with a final member of the set.

Example 1.1.1. The set of integers between 1 and 100 that are divisible by 3 can be expressed with informal list notation as $\{3, 9, 12, \ldots, 99\}$.

1.1.1 Set cardinality

The **cardinality** of set A, denoted $|A|$, is equal to the number of members of A. A set may be classified based on its type of cardinality.

Finite a set of the form $A = \{s_1, s_2, \ldots, s_n\}$, in which case $|A| = n$

Countably Infinite a set whose members can be written in an infinite list, with no members being repeated, in which case $|A| = \infty$

Uncountably Infinite a set that is not finite and whose members cannot be written in an infinite list having no repeats

Example 1.1.2. The set of **natural numbers** $\mathcal{N} = \{0, 1, 2, \ldots\}$ is countably infinite since its members may be listed as $0, 1, 2, \ldots$. Set of integers $\mathcal{I}$ is also countably infinite, since its members may be listed as $0, 1, -1, 2, -2, \ldots$. □

Example 1.1.3. A rational number is any number that may be written as a fraction p/q, where p and q are integers, and $q \neq 0$. Perhaps somewhat surprising is that the set $\mathcal{Q}$ of rational numbers is countably infinite. To show this we use a method called **dovetailing**, where members of $\mathcal{Q}$ are listed in rounds. In particular, i members of $\mathcal{Q}$ are listed in round i, $i = 1, 2, \ldots$. In what follows we show how the positive rationals can be listed with dovetailing. In round 1 we list $1/1$. In round 2 we list $1/2, 2/1$. In round 3 we list $1/3, 2/2, 3/1$. In round 4 we list $1/4, 2/3, 3/2, 4/1$. See the pattern? To generalize, in round i we list $1/i, 2/(i-1), 3/(i-2), \ldots, i/1$. We leave it as exercise to compute the round for which fraction p/q is listed, for positive integers p and q. Finally, notice that in odd rounds the number 1 is always listed. For example, in round 1 it is listed as $1/1 = 1$, and in round 3 as $2/2 = 1$. Since our infinite list is not allowed to have repeats, before listing a fraction we must first check if it has already been listed (in the form of a different fraction) in a previous round. If yes, then we do not list it again. Therefore, we've demonstrated a way to list all rational numbers in a way that each number appears exactly once in the list. □

For the remainder of the book, $\mathcal{N}$ denotes the natural numbers $\{0, 1, 2, \ldots\}$, $\mathcal{I}$ the integers, $\mathcal{Q}$ the rational numbers, and $\mathcal{R}$ the real numbers. Also, the addition of a plus, such as $\mathcal{R}^+$, denotes the positive members of a numerical set, while the addition of a minus, such as $\mathcal{I}^-$ denotes the negative members of a numerical set.

1.1.2 Subsets

Set A is said to be a **subset** of set B iff every member of A is also a member of B. This is symbolically denoted by $A \subseteq B$. Moreover, A is said to be a **proper subset** of B, denoted $A \subset B$, iff it is a subset of B, but there is some member of B who is not a member of A.

Example 1.1.4. Sets $\{3\}$ and $\{1, 3\}$ are proper subsets of $\{1, 2, 3\}$, while $\{1, 2, 3\}$ is a subset of $\{1, 2, 3\}$, but not a proper subset. Also, is $A = \{\{1, 2\}, \{3, 4\}, 5\}$ a subset of $B = \{1, 2, 3, 4, 5\}$? No since, e.g., $\{1, 2\} \in A$ (yes, a set can be a member of another set!), but $\{1, 2\} \not\in B$. This is true since $\{1, 2\}$ is a set, and B's members are the natural numbers 1 through 5.

The **power set** of a set S, denoted $\mathcal{P}(S)$, is defined as the set of all subsets of set S.

Example 1.1.5. For $S = \{1, 2, 3\}$ we have

$$\mathcal{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Proposition 1.1.1. If S is a finite set with $|S| = n$, then $|\mathcal{P}(S)| = 2^n$.

Proof of Proposition 1.1.1. Without loss of generality, we may assume that $S = \{1, 2, \ldots, n\}$. Let A be a subset of S and consider the statements $i \in A$, for $i = 1, 2, \ldots, n$. Each of these statements will either evaluate to true (1) or false (0), depending on the choice of i and A. For example, if $A = \{1, 3, 5\}$ and $i = 2$, then the statement $2 \in A$ evaluates to 0, since 2 is not a member of A. Placing these statement evaluations side-by-side, we get the binary sequence

$$(1 \in A, 2 \in A, \ldots, n \in A).$$

For example, if $n = 5$ and $A = \{1, 3, 5\}$, then

$$(1 \in A, 2 \in A, 3 \in A, 4 \in A, 5 \in A) = (1, 0, 1, 0, 1).$$

Conversely, every binary sequence of length n corresponds with a subset of S. For example, if $n = 5$, then $(0, 0, 1, 0, 0)$ corresponds with subset $\{3\}$, while $(1, 1, 1, 0, 1)$ corresponds with $\{1, 2, 3, 5\}$. Therefore, the number of subsets of S is equal to the number of binary sequences having length n, and the reader may check that there are 2^n such sequences. As an example, the following **subset table** shows the correspondence between binary sequences of length three and subsets of $S = \{1, 2, 3\}$.

$1 \in A$	$2 \in A$	$3 \in A$	Subset A
0	0	0	$\emptyset$
0	0	1	$\{3\}$
0	1	0	$\{2\}$
0	1	1	$\{2, 3\}$
1	0	0	$\{1\}$
1	0	1	$\{1, 3\}$
1	1	0	$\{1, 2\}$
1	1	1	$\{1, 2, 3\}$

$\square$

1.1.3 Set operations

A binary arithmetic operation represents a way of taking two numbers, a and b, and associating with them a third number c that is obtained by performing the operation on a and b. For example, the binary addition operation associates with a and b the number $c = a + b$. In what follows we introduce some analogous binary set operations. Each of these operations takes two sets, A and B, and associates with them a third set in accordance with some rule of operation.

Union $x \in A \cup B$ iff either $x \in A$ or $x \in B$

Intersection $x \in A \cap B$ iff $x \in A$ and $x \in B$

Difference $x \in A - B$ iff $x \in A$ and $x \notin B$

Symmetric Difference $x \in A \oplus B$ iff $x \in A$ or $x \in B$, but not both; i.e. $A \oplus B = (A - B) \cup (B - A)$

Cartesian Product $(a, b) \in A \times B$ iff $a \in A$ and $b \in B$

Notice that for the Cartesian product, a member of $A \times B$ has the form (a, b), which is called a (binary) **tuple**, while a and b each represent a **component** of the tuple. Note that the term "tuple" is synonymous with several others in math and computing, including *array*, *vector*, *list*, and *sequence*. These tuples of $A \times B$ represent all possible ways of combining a member of A with a member of B. Also, the Cartesian product may be generalized to an arbitrary number of sets, such as $A_1 \times A_2 \times \cdots \times A_k$, in which case the members of this set are k-tuples of the form $(a_1, a_2, \ldots, a_k)$, such that $a_i \in A_i$, for all $i = 1, 2, \ldots, k$.

Example 1.1.6. Given sets $A = \{1, 3, 5, 6, 7, 9\}$ and $B = \{0, 2, 4, 5, 6, 7, 8, 10\}$, we have $A \cup B = \{0, 1, 2, \ldots, 10\}$, $A \cap B = \{5, 6, 7\}$, $A - B = \{1, 3, 9\}$, and $A \oplus B = \{0, 1, 2, 3, 4, 8, 9, 10\}$.

Example 1.1.7. Given sets $A = \{a, b\}$, $B = \{1, 2, 3\}$, then $A \times B = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3)\}$.

In addition to the above binary operations, there one important unary operation called the **complement** of set A, and denoted $\overline{A}$. In particular, $x \in \overline{A}$ iff $x \notin A$. The complement is only useful in cases when there is a pre-defined **universe** $\mathcal{U}$ consisting of all possible items that could appear in any set. In this case we have $\overline{A} = \mathcal{U} - A$. For example, if the universe $\mathcal{U}$ is given as the integers from 1 to 10, and $A = \{1, 3, 4, 8\}$, then $\overline{A} = \{1, 2, 5, 7, 9, 10\}$.

We say that two sets A and B are **disjoint** iff $A \cap B = \emptyset$. For example, the set of even integers and the set of odd integers are disjoint, since no integer is both even and odd.

The following is a table of identities that are always true for any three sets A, B, and C.

Identity	**Description**
$A \cup \emptyset = A$ $A \cap \mathcal{U} = A$	Identity
$A \cup \mathcal{U} = \mathcal{U}$ $A \cap \emptyset = \emptyset$	Domination
$\overline{\overline{A}} = A$	Double Complement
$A \cup A = A$ $A \cap A = A$	Idempotent
$A \cup B = B \cup A$ $A \cap B = B \cap A$	Commutative
$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$	Associative
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (B \cap C)$	Distributive
$\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$	De Morgan

1.2 Variables

A **variable** is an entity that has a name and a set, called the **domain** of the variable. In this book we require that a variable's name begin with a lowercase letter. Moreover, lowercase letters $v, w, x, y, z, \ldots$ are used when denoting an *abstract variable*, meaning some arbitrary variable whose name and domain have not been specified. With that said, given some variable v, $\mathrm{dom}(v)$ denotes its domain set. For example, if a variable named `amount` has domain $\{1, 2, 3, 4\}$, then we may write $\mathrm{dom}(\texttt{amount}) = \{1, 2, 3, 4\}$.

Variables are essential to mathematics and problem solving in general. Indeed, finding and verifying a solution to a problem means sifting through a sea of possibilities and eventually identifying an entity (be it a number, tuple, policy, algorithm, etc.) that allows for some issue to be resolved in a way that satisfies a set of constraints, where each constraint may be viewed as a criterion that must be met by any solution.

As an example, suppose the solution we seek is a number between 1 and 1000, and that one constraint is that the number should be at least seven. By representing the solution with a variable, say x with $\texttt{dom}(x) = \{1, 2, \ldots, 1000\}$, we may then write the above constraint as $x \geq 7$. Not only is this abstract representation more succinct, but it also makes it possible to find a solution via mathematics and/or some automated computing procedure.

Given variable v, an **assignment** to v is some $d \in \mathrm{dom}(v)$ that is substituted for v whereever v may appear in some constraint. For example, if variable x has domain $\{1, 2, \ldots, 1000\}$, and we have the constraint $x \geq 7$, then an assignment of $d = 9$ to x allows us to substitute 9 for x and obtain the constraint $9 \geq 7$. We say that this assignment **satisfies** the constraint, since it produces the statement $9 \geq 7$ which is a true statement.

One strength of constraint programming is that it allows for a variable domain to be arbitrarily complex. For example, suppose we need ten trucks to deliver five hundered packages in a city in the least amount of time. Then the solution we seek is a ten tuple t, where each component of t is a driving route that begins and ends at the package terminal. Therefore, the domain of t consists of all such possible tuples. In a later chapter we provide a means for defining such a domain.

1.3 Propositional Logic

By a constraint, we mean something that places limitations on what can represent a solution to a given problem. Moreover, a constraint is usually expressed in the form of a *rule*, a word that is synonymous with constraint. For example, the statement "Ronald only studies calculus on weekdays" represents a constraint that limits when Ronald can study. So if Ronald belongs to a calculus study group that only meets during times that are convenient to every member (another constraint!), then we can conclude that this group never meets on the weekends. Notice how rules stated like the one above lend themselves to being evaluated as being true or false. For this reason such a rule is referred to as a **logical proposition**, and is capable of being modeled using a form of logic known as **propositional logic**. For example, in propositional logic the statement made about Ronald may be expressed using a single **Boolean variable** named s, where $\texttt{dom}(s) = \mathcal{B} = \{\texttt{true},\texttt{false}\}$ is the set consisting of the Boolean values `true` and `false`. In other words, s may be assigned the value of `true` or `false`, since we believe that either Ronald does in fact only study calculus on the weekdays, or that there has been or will be an occurrence of Ronald studying calculus on the weekend. There is no "middle ground". Of course, there could be some gray area in terms of what it means to "study calculus". Does this include solving an interesting calculus problem on a Saturday morning, just for fun? Or how about mentally reviewing the definition of the Chain Rule of calculus just before bed on Sunday? Nevertheless, we assume that our definition of "studying calculus" allows us to answer these questions.

A single Boolean variable v is referred to as an **atomic proposition**, since it does not reduce further to other more basic propostions. On the other hand, a **compound proposition** is one that can be expressed using one or more atomic propositions, together with one or more *logical connectives.* For example, the statement "I'm either having the miso soup, or the tofu green salad" represents a compound proposition, since it uses logical exclusive-OR to connect two atomic propositions: "I'm having the miso soup", and "I'm having the tofu green salad". Here, exclusive-OR means that one of the statements must be true, but not both, since we assume that the speaker is implicitly indicating that she plans to select only one of the two dishes as part of her meal.

The exclusive-OR logical connective is just one of several that we now present in the following table.

Connective	**Symbol**	**Example**
NOT	$\neg$	"It is **not** raining outside."
AND	$\wedge$	"It is raining **and** the sidewalk is wet."
OR (Inclusive)	$\vee$	"A wrench **or** pliers is needed for the repair."
OR (Exclusive)	$\oplus$	"He was born in either Phoenix **or** Tuscon."
IF-THEN	$\rightarrow$	"**If** it rains **then** the sidewalk gets wet."
EQUIVALENCE	$\leftrightarrow$	"You pass **if and only if** you study."

The NOT connective acts on a single proposition, and asserts its falsehood. Each of the remaining connectives combines two propositions and asserts something in relation to the two. Indeed, AND asserts that both are true, inclusive-OR asserts that one of the two (or both) are true, exclusive-OR asserts that one of the two (but not both) are true, IF-THEN asserts that the first being true is a cause for the second to be true, and EQUIVALENCE asserts that either both are true, or both are false. Thus, with the exception of IF-THEN, the order in which the two propositions appear does not matter. For example, saying that "A wrench or pliers is needed for the repair" has the same truth as "Pliers or a wrench is needed for the repair" (although in practice one may want to first state the tool of preference). On the other hand, the statement "If the sidewalk is wet, then it is raining" has a different meaning than "If it is raining then the sidewalk is wet", since the former is stating that the wetness of the sidewalk is enough to guarantee rain, which is not always true (think of sprinklers).

1.3.1 Truth tables

Like the set operations defined in the previous section, we may think of each binary logical connective as a means of taking two Boolean values a and b, and associating with them a third Boolean value c that is obtained by applying the connective to a and b. For example, if $a = \texttt{false}$ and $b = \texttt{true}$, then $a \wedge b = \texttt{false}$, since it is false that both a and b are true. For this reason each of the logical connectives is referred to as a **Boolean operation**, with NOT being a unary operation, and all others being binary operations. A unary Boolean operation only has two possible input values, namely `false` and `true`, while a binary operation has $2 \times 2 = 4$ different possible input combinations, namely

$$(\texttt{false}, \texttt{false}), (\texttt{false}, \texttt{true}), (\texttt{true}, \texttt{false}), (\texttt{true}, \texttt{true}).$$

Since there are only a handful of possible input combinations, we may show in a table the output of a Boolean operation in response to each of the input combinations. Such a table is referred to as a **truth table**.

For example, the following is a truth table for NOT.

p	$\neg p$
0	1
1	0

Notice that **false** is represented by 0 and **true** by 1. The first column gives the two possible truth values for some logical proposition p, while the second column gives the corresponding truth value of $\neg p$. This next truth table is for AND.

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Here the first two columns provide the different truth combinations for the input propositions p and q, while third column gives the corresponding truth value of $p \wedge q$. Notice that $p \wedge q$ can only evaluate to 1 when both p and q evaluate to `true`. Table 1.1 shows the truth tables of the remaining Boolean operations combined into one table.

The $p \to q$ truth table seems worthy of comment. Recall from above that $p \to q$ is asserting that the truth of p is a cause for the truth of q. Here we refer to p as the **cause** and q as the **effect**. For example, if p represents "It's raining" and q represents "The sidewalk is wet", then $p \to q$ asserts that rain is a cause for a sidewalk getting wet. But it does *not* assert that rain is *the* cause for a sidewalk to get wet. Indeed, there could be no

p	q	$p \vee q$	$p \oplus q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	1	1
0	1	1	1	1	0
1	0	1	1	0	0
1	1	1	0	1	1

Table 1.1: Truth tables for some Boolean operations

rain, but the sprinklers are causing the sidewalk to get wet. Thus $0 \rightarrow 1$ evaluates to `true`, since a wet sidewalk on a sunny day only establishes that there is more than one cause for a wet sidewalk, and in no way negates the fact that rain is one of the causes. Similarly, $0 \rightarrow 0$ evaluates to `true` since a dry sidewalk on a sunny day does nothing to disprove the causal link between between rain and wet sidewalks. In a system of logic that allows for more than two possible truth values, perhaps $0 \rightarrow 1 = 1$ would not be the preferred evaluation. But with only two choices, `true` or `false`, `false` seems too harsh a choice, and so we are left with `true`. An IF-THEN statement that is true because the cause is false is said to be **vacuously** true.

It's worth emphasizing that the p and q shown in the above truth tables can represent either atomic or compound propositions. For example, consider the compound propositional formula $(a \wedge b) \rightarrow c$. It has the form $p \rightarrow c$, where p is the **subformula** $a \wedge b$, i.e. a formula that occurs within the main formula. Now we could make a truth table for $(a \wedge b) \rightarrow c$ using the variables p and c, but it would look exactly like the one in Table 1.1, and would obfuscate the truth of the formula in relation to the original three variables a, b, and c. Moreover, there are a total of $2 \times 2 \times 2 = 8$ different truth-value combinations that can be assigned to these input variables, and so we need a table with eight rows. Table 1.2 shows the desired truth table. Notice that the table includes an additional column that gives the truth values for $(a \wedge b)$. This column serves as the "cause" column, while the c-column serves as the "effect" column which are both needed to compute

a	b	c	$a \wedge b$	$(a \wedge b) \rightarrow c$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Table 1.2: Truth table for $(a \wedge b) \rightarrow c$

the $((a \wedge b) \rightarrow c)$-column. For example, row 7 of the $((a \wedge b) \rightarrow c)$-column evaluates to 0 because the cause value in that row is 1 while the effect value is 0, giving $1 \rightarrow 0 = 0$.

1.3.2 Logical equivalence

Definition 1.3.1. Given propositional formula p, then p is a

1. **tautology** iff p always evaluates to `true`, regardless of how its variables are assigned.
2. **fallacy** iff p always evaluates to `false`, regardless of how its variables are assigned.
3. **contingency** iff p's variables may be assigned in a way that makes p evaluate to `true`, and may be assigned in another way that makes p evaluate to `false`.

Example 1.3.1. Let a and b be Boolean variables. Then

a	b	$b \to a$	$a \to (b \to a)$
0	0	1	1
0	1	0	1
1	0	1	1
1	1	1	1

Table 1.3: Truth table for $a \to (b \to a)$

1. a is a contingency since $a = 0$ makes a false, and $a = 1$ makes a true.
2. $a \to b$ is a contingency since $(a = 1, b = 0)$ makes $a \to b$ false, while $(a = 1, b = 1)$ makes $a \to b$ true.
3. $a \lor \neg a$ is a tautology since, no matter how a is assigned, either a or $\neg a$ will evalute to `true`.
4. $a \land \neg a$ is a fallacy since, no matter how a is assigned, either a or $\neg a$ will evalute to `false`, and so a and $\neg a$ cannot both be true.
5. $a \to (b \to a)$ is a tautology. One way to see this is by constructing a truth table and verifying that the $a \to (b \to a)$-column consists of all 1's (see Table 1.3). Alternatively, notice that the only way this formula could be false is if a is true and $b \to a$ is false. But $b \to a$ is false only when a is false, which conflicts with the requirement that a be true. Therefore, the formula must be tautology.
6. $(a \to b) \to (\neg a \to \neg b)$ is a contingency, since it's true for $(a = 0, b = 0)$, but false for $(a = 0, b = 1)$.

Two propositional formulas p and q are said to be **logically equivalent** iff the formula $p \leftrightarrow q$ is a tautology. In other words, q is true whenever p is true, and p is true whenever q is true. The most common way to show that two formulas are logically equivalent is to construct their respective truth tables and verify that the p-column of p's table is identical with the q-column of q's table.

a	b	$a \to b$	$\neg b$	$\neg a$	$\neg b \to \neg a$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	0	0
1	1	1	0	0	1

Table 1.4: Truth tables for $a \to b$ and $\neg b \to \neg a$

Example 1.3.2. Proposition $a \to b$ is logically equivalent to $\neg b \to \neg a$. One way of seeing this is to construct a truth table for both formulas (see Table 1.4) and verify that the $(a \to b)$-column is identical to the $(\neg b \to \neg a)$-column. Alternatively, notice that $a \to b$ can only be made false by the assignment $(a = 1, b = 0)$. Similarly, $\neg b \to \neg a$ can only be made false when $\neg a$ is false and $\neg b$ is true, i.e. can only be made false by the assignment $(a = 1, b = 0)$. Therefore, they are equivalent.

Table 1.5 shows the most important logical equivalences of propositional logic. These equivalences are also often referred to as *logical axioms* and *logical identities*. For the sake of readability, 1 represents `true`, 0 represents `false`, and $=$ represents $\leftrightarrow$. Notice that each equivalence has a corresponding **dual** equivalence that is obtained by replacing $\vee$ with $\wedge$ (or $\wedge$ with $\vee$) and 1 with 0 (or 0 with 1). For example, the dual of $p \vee 1 = 1$ is $p \wedge 0 = 0$. Notice also that these equivalences match the set identities from Section 1.1, with $\vee$ corresponding to $\cup$, $\wedge$ to $\cap$, 1 to $\mathcal{U}$, and 0 to $\emptyset$. In the parlance of mathematics we say that, together with their respective operations, the set $\mathcal{L}$ of all logical propositions and the set $\mathcal{S}$ of sets are both examples of a mathematical structure called a **Boolean algebra**, and Table 1.5 lists the axioms (expressed in logical form) of such an algebra.

Equivalence	**Name**
$p \wedge 1 = p$ $p \vee 0 = p$	Identity
$p \vee 1 = 1$ $p \wedge 0 = 0$	Domination
$p \vee p = p$ $p \wedge p = p$	Idempotency
$\neg(\neg p) = p$	Double negation
$p \vee q = q \vee p$ $p \wedge q = q \wedge p$	Commutativity
$(p \vee q) \vee r = p \vee (q \vee r)$ $(p \wedge q) \wedge r = p \wedge (q \wedge r)$	Associativity
$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	Distributivity
$\neg(p \vee q) = \neg p \wedge \neg q$ $\neg(p \wedge q) = \neg p \vee \neg q$	De Morgan

Table 1.5: The most commonly used logical equivalences

1.3.3 More on conditional propositions

A proposition of the form $p \to q$ is called a **conditional** proposition, and appears quite often within natural language. When p and q are often respectively referred to as the **cause** and **effect**, or the **hypothesis** and **conclusion**.

Definition 1.3.2. Given the proposition $p \to q$,

1. $q \to p$ is called its **converse**
2. $\neg q \to \neg p$ is called its **contrapositive**
3. $\neg p \to \neg q$ is called its **inverse**

Example 1.3.3. Given the statement "If it's raining, then the sidewalk is wet", we have $p =$ "it's raining" and $q =$ "the sidewalk is wet". Then the converse reads "If the sidewalk is wet then it's raining", the contrapositive reads "if the sidewalk is dry, then it's not raining", while the inverse reads "if it's not raining, then the sidewalk is dry".

Proposition 1.3.1. The proposition $p \to q$ is logically equivalent to its contrapositive, while its converse is logically equivalent to its inverse.

Proof of Proposition 1.3.1. The logical equivalence of $p \to q$ and $\neg q \to \neg p$ was demonstrated in Example 1.3.2. Finally, the converse $q \to p$ is logically equivalent to the inverse $\neg p \to \neg q$, since the inverse is the contrapositive of the converse. $\square$

The following are some of the most common ways that a conditional statement $p \to q$ may be expressed in English.

p **implies** q**:** "Tingling of the tongue implies a vitamin B-12 deficiency."

if p**,** q**:** "If you choose not to decide, you still have made a choice."

p **only if** q**:** "You will pass this class only if you study."

p **is sufficient for** q**:** "Scoring an A on all exams is sufficient for receiving an A in the class."

q **if** p**:** "A number n is prime if it is not divisible by a number between 2 and $\lfloor\sqrt{n}\rfloor$."

q **whenever** p**:** "We will leave whenever Iqbal arrives."

q **is necessary for** p**:** "Studying outside of class at least five hours a week is necessary for passing this class."

Of the above examples, perhaps the two most misunderstood are " p only if q" and "q is necessary for p". Both of these have essentially the same meaning. For example, "p only if q" means that the only way p can be true is if q is true. In other words, if q is false, then so is p, i.e. $\neg q \rightarrow \neg p$ which is logically equivalent to $p \rightarrow q$. Thus, if $p =$ "you pass the class" and $q =$ "you study", then the statement "You will pass this class only if you study" is indicating that studying is not necessarily sufficient for passing, but if you pass the class, then certainly you studied. This may seem odd, since studying is the "effect" even though it precedes passing the class on the timeline. For this reason, it seems more natural to write the statement as "studying is necessary for passing the class".

1.4 Functions

Functions are considered the workhorses of any programming language, because they often serve the purpose of transforming data into other (perhaps more useful) data. For example, suppose that on a given day a package-delivery company has 97 working delivery trucks, 136 able and willing drivers (each with a unique preferred work schedule), and 8,643 packages that need delivering by various times and to various residential and business addresses throughout some region. What this company needs is a function that can input this data, and output a delivery route for each truck, a work schedule for each driver, and an assigned truck for each driver so that all the packages are delivered on time. Also, functions form the building blocks of expressions, which in turn can be used to create formal languages, such as the one used in this book.

Let x be a variable whose domain set is A, and let B be a set. Then a **function** f of a single variable x into B represents a means for assigning every $a \in A$ with exactly one $b \in B$. The following is some terminology and notation that is used to describe functions.

- A is called the **domain** of f.
- B is called the **codomain** of f.
- $f : A \to B$ is a notation that indicates f is a function that assigns members of set A to members of set B.
- $f(x)$ is a common notation that is used when both the domain of x (and hence f) and the codomain of f are already understood.
- $f(a) = b$ indicates that f has assigned $a \in A$ to $b \in B$, where b is called the **image** of a under f, and a is called a **preimage** of b under f. It is also common to call a the **input** and $f(a) = b$ its assigned **output**.

Like variables, every function has a name, and it is common to use generic names, such as f, g, and h, when speaking of some arbitrary (i.e. no one in particular) function, just as it is common to use names like x, y, and z to represent an arbitrary variable.

Example 1.4.1. Consider the function grade(s) that assigns student s a letter grade from the set $G = \{a, b, c, d, f\}$, where

$$\mathrm{dom}(s) = S = \{\mathrm{Ann}, \mathrm{Ethan}, \mathrm{Jaspinder}, \mathrm{Pam}\}.$$

Thus, S is the domain of `grade`, and G is its codomain. Finally the student grades assigned by `grade` are grade(Ann) $= a$, grade(Ethan) $= c$, grade(Jaspinder) $= c$, and grade(Pam) $= b$.

Example 1.4.2. Let S denote the set of students attending a university, I the set of instructors at the university, and s a variable whose domain is S. For a student s attending a university, let ins(s) denote the instructor of the first calculus course that s enrolled in at the university. Then ins : $S \to I$ is not a function because not every student (e.g. a student majoring in music) will elect to take calculus at the university. A function must assign *every* member of the domain to some member of the codomain. $\square$

Example 1.4.3. Consider the following description of a function $f(x)$. For positive real number x, $f(x)$ is the real number whose square equals x. Is this a valid function? On the surface it may appear that way, but consider $x = 4$. From the description $f(x)$ is the real number whose square equals 4. There are actually two such numbers: 2 and -2, but it's not valid to say $f(4) = \pm 2$ since, by definition, a function assigns each input x to *exactly* one member in the codomain which in this case is $\mathcal{R}$. On the other hand, had the description read "$f(x)$ is the *positive* real number whose square equals x", then there is no ambiguity about the output, namely $\sqrt{x}$, to which x is being assigned. In mathematics, a **relation** is a structure that allows an input to be assigned to more than one output. $\square$

1.4.1 Describing a function

When the domain of a function $f(x)$ is finite and not too large, then the most common way of describing f is to list each $f(a) = b$ pair, as was done in Example 1.4.1. On the other hand, when the domain is infinite or very large, then it becomes necessary to use a rule to describe the relationship between the input and output. Such rules are usually algebraic, in that they are described with an algebraic expression that depends on the input variable x. For example, if x has the domain of all real numbers, then $f(x) = 3x^2 + 7x - 5$ provides a rule for assigning an arbitrary domain input to a codomain output. Indeed, x represents an arbitrary domain input, and the rule states that x should be assigned to codomain value $3x^2 + 7x - 5$. For example, if $x = -2$, then $f(-2) = 3(-2)^2 + (7)(-2) - 5 = -7$. This is an example of a **real-valued function**, meaning that both its domain and codomain is equal to the set of real numbers $\mathcal{R}$. In fact, the main goal of calculus is to study the properties of real-valued functions.

Example 1.4.4. The **floor** operation, denoted $\lfloor x \rfloor$ appears often in function rules, where $\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to real number x. Some examples include $\lfloor 3.2 \rfloor = 3$, $\lfloor 7 \rfloor = 7$, $\lfloor 0.341 \rfloor = 0$, and $\lfloor -2.45 \rfloor = -3$. Similarly, the **ceiling** operation, denoted $\lceil x \rceil$ is defined as the least integer that is greater than or equal to x. Some examples include $\lceil 3.2 \rceil = 4$, $\lceil 7 \rceil = 7$, $\lceil 0.341 \rceil = 1$, and $\lceil -2.45 \rceil = -2$. $\square$

In practice, many functions require far more complex rules than what can be provided by a single algebraic expression. Such rules are usually described with a procedural programming language, such as Python or C++. Although we do not study this approach here, in a later chapter we describe how to use a *constraint* program to define the relationship between a function's input and output.

1.4.2 Properties of functions

The **range** of a function $f : A \to B$, denoted range(f) is the set of all outputs of function f. Thus, range(f) $\subseteq B$.

Example 1.4.5. For the `grade` function from Example 1.4.1, we see that range(grade) $= \{a, b, c\}$ since one student was assigned an a, one a b, and two were assigned c. On other hand no one was assigned d or f, and so $d, f \not\in$ range(grade). Similarly, range($\lfloor x \rfloor$) $= \mathcal{I}$, since the floor function only outputs integers, and every integer serves as an output for some real input, e.g. $\lfloor n \rfloor = n$ for every $n \in \mathcal{I}$.

Properties of functions: let $f : A \to B$ be a function. Then f is

one-to-one iff for every $b \in B$ there is at most one $a \in A$ such that $f(a) = b$.

onto iff for every $b \in B$ there is at least one $a \in A$ such that $f(a) = b$.

one-to-one correspondence iff it is both one-to-one and onto.

Thus, a function $f : A \to B$ is one-to-one provided every member $b \in B$ has at most one domain member assigned to it, and f is onto if and only if range(f) $= B$.

Figures 1.4.1 and 1.4.2 show examples of one-to-one and onto functions, respectively. For example, an arrow from the node labeled with a to the node labeled with 2 means that $f(a) = 2$.

Example 1.4.6. Consider the function $f : \mathcal{N} \to S$, where S is the set of all bit strings, such as 0, 100, 010111, etc.. Moreover, $f(n) = s$, where s is

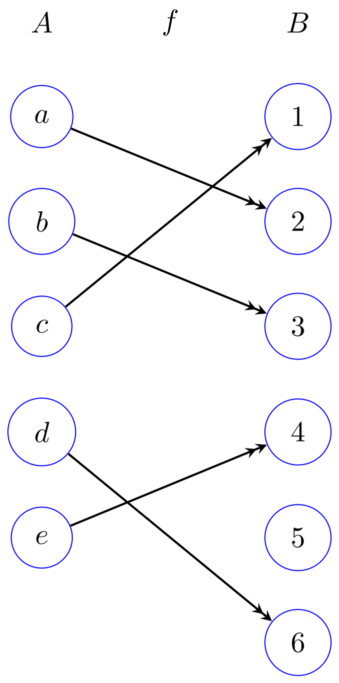

Figure 1.4.1: Example of a one-to-one function

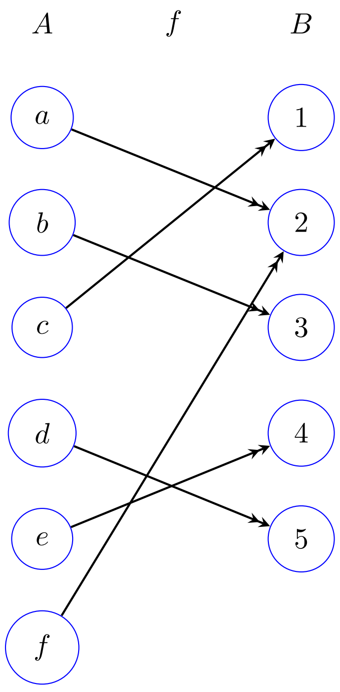

Figure 1.4.2: Example of an onto function

the shortest bit string that represents n as a binary number. For example, $f(6) = 110$ since $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6$. First notice that f is a well-defined function since every natural number can be written in binary, and there is a unique smallest bit string whose decimal value equals n. Also, f is one-to-one since every bit string s has a unique decimal value, say n, and so *at most* one natural number, namely n, can be assigned to s. However, f is not onto since not every bit string has a preimage. For example, $s = 00011001$ has an associated decimal value of $n = 25$, but $f(25) = 11001 \neq 00011001$, since 11001 and 00011001 are different strings. $\square$

Example 1.4.7. Consider the previous example, but now interchange the domain and codomain, meaning that we now have a function $f : S \to \mathcal{N}$, where f assigns each bit string s the integer n, where s is a valid binary representation of n. For example, $f(001101) = 13$, since

$$13 = 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Again, notice that f is well-defined, since every bit string is a binary representation for some unique $n \in \mathcal{N}$. Secondly, f is also onto, since every $n \in \mathcal{N}$ has a preimage s, i.e. $f(s) = n$, since every $n \in \mathcal{N}$ has a binary representation. Finally, f is *not* one-to-one, since every $n \in \mathcal{N}$ has infinitely many preimages. In fact if $f(s) = n$, then $f(0^m \cdot s) = n$ as well, where $0^m \cdot s$ is the bit string obtained by placing m zeros in front of s, for any $m \geq 0$. On the other hand, being one-to-one means that every codomain member is allowed at most one preimage. $\square$

1.4.3 Images and preimages of sets

Let $f : A \to B$ be a function. Then for $b \in B$, $f^{-1}(b) \subseteq A$ denotes the set of preimages of b with respect to f.

Example 1.4.8. For the floor function $f(x) = \lfloor x \rfloor$, $f^{-1}(2)$ denotes the real numbers x for which $\lfloor x \rfloor = 2$. But this is all real numbers between 2 and

3, not including 3. Thus, using interval notation from calculus, we have $f^{-1}(2) = [2, 3)$, where the left bracket indicates that 2 is included in the set, while the right parenthesis indicates that 3 is not included. Similarly, for $f(x) = \lceil x \rceil$, $f^{-1}(2) = (1, 2]$, where the left parenthesis indicates that 1 is not in the set, since $f(1) = 1 \neq 2$.

Example 1.4.9. For the function $f : S \to \mathcal{N}$ from Example 1.4.7, we have

$$f^{-1}(6) = \{110, 0110, 00110, \ldots\}$$

which is the set of all bit strings that begin with m 0's, followed by 1100, where $m \geq 0$.

Given $f : A \to B$, now let $S \subseteq A$ and $T \subseteq B$ be subsets of the domain and codomain respectively. We have the following definitions.

f-image of S denoted as $f(S)$, and is defined as the union of all images $f(s)$, where $s \in S$ is arbitrary.

f^{-1}-preimage of T denoted as $f^{-1}(T)$, and is defined as the union all of preimages $f^{-1}(t)$, where $t \in T$ is arbitrary.

Example 1.4.10. For the function $f(x) = \lfloor x \rfloor$, $f([-3.5, 2.8])$ is the set of all images of real numbers between -3.5 and 2.8. But the floor of all of these numbers falls between -4 and 2. Therefore, $f([-3.5, 2.8]) = \{-4, -3, -2, -1, 0, 1, 2\}$.

Example 1.4.11. For the function $f : \mathcal{N} \to S$ from Example 1.4.6, $f^{-1}(\{00, 01, 10, 11\})$ denotes set of natural numbers n for which $f(n) \in \{00, 01, 10, 11\}$. And there are only two such n, namely 2 and 3 (why?). Therefore, $f^{-1}(\{00, 01, 10, 11\}) = \{2, 3\}$.

1.4.4 Operations on functions

Let f and g be two functions, both having the same domain A and codomain B. Notice that whatever operations that may be performed on members of B may also be performed on f and g. For example, suppose $B = \mathcal{R}$. In this case we may perform arithmetic operations on members of B, since these members are real numbers. But this in turn allows us to perform arithmetic operations on f and g. For example, since we may add two real numbers b_1 and b_2 to get a third real number $b_1 + b_2$, so we may add f and g to get a third function, $f + g$, whose domain is A, codomain is $B = \mathcal{R}$, and whose rule is $(f + g)(a) = f(a) + f(b)$. The following definition summarizes the most common arithmetic operations on functions.

Definition 1.4.1. Let $f, g : A \to B$ be functions, and suppose $B \subseteq \mathcal{R}$ is a set of real numbers that is closed under addition, subtraction, multiplication, and division (e.g. B is closed under addition iff $b_1, b_2 \in B$ implies $b_1 + b_2 \in B$). Then the following functions are well-defined for all $a \in A$.

Addition $(f + g)(a) = f(a) + g(a)$

Subtraction $(f - g)(a) = f(a) - g(a)$

Multiplication $(fg)(a) = f(a)g(a)$

Division $(f/g)(a) = f(a)/g(a)$ (Exception: $(f/g)(a)$ is undefined whenever $g(a) = 0$)

Example 1.4.12. Let f and g be real-valued functions having rules $f(x) = 2x + 5$, and $g(x) = x^2 - 3$. Then the rules for $f + g$, $f - g$, fg, and f/g are $(f + g)(x) = (2x + 5) + (x^2 - 3) = x^2 + 2x + 2$,

$$(f - g)(x) = (2x + 5) - (x^2 - 3) = -x^2 + 2x + 8,$$

$$(fg)(x) = (2x + 5)(x^2 - 3) = 2x^3 + 5x^2 - 6x - 15,$$

and $(f/g)(x) = (2x+5)/(x^2-3)$. Futhermore, since $f(2) = (2)(2)+5 = 9$, and $g(2) = 2^2-3 = 1$, we have $(f+g)(2) = 9+1 = 10$, $(f-g)(2) = 9-1 = 8$, $(fg)(2) = (9)(1) = 9$, and $(f/g)(2) = 9/1 = 9$. Of course, we should get the same answers by using the rules for $f+g, \dots, f/g$. For example, $(f+g)(2) = 2^2+2(2)+2 = 10$. We leave it to the reader to verify this for $f-g, fg$, and f/g. $\square$

In a similar manner to how arithmetic operations were defined for functions, we may also define set operations on functions having codomains that are closed under set operations. In what follows we let $\mathcal{S}$ denote the set of all possible sets (notice how $\mathcal{S}$ is a member of itself!).

Definition 1.4.2. Let $f, g : A \to B$ be functions, and suppose $B \subseteq \mathcal{S}$ is a set of sets that is closed under union, intersection, subtraction, symmetric difference, and complement. Then the following functions are well-defined for all $a \in A$.

Union $(f \cup g)(a) = f(a) \cup g(a)$

Intersection $(f \cap g)(a) = f(a) \cap g(a)$

Subtraction $(f - g)(a) = f(a) - g(a)$

Symmetric Difference $(f \oplus g)(a) = f(a) \oplus g(a)$

Complement $\overline{f}(a) = \overline{f(a)}$

Example 1.4.13. Let $A = \mathcal{N}$, and B denote the power set of all prime numbers. Then $f : A \to B$ is defined so that $f(n)$ is the set of all prime factors of n; i.e. those prime numbers that divide evenly into n. For example, since $24 = 2^3 \cdot 3^1$, we have $f(24) = \{2, 3\}$. Also, let $g : A \to B$ be defined so that $g(n)$ is the set of prime numbers p for which $\sqrt{n} \le p \le n$. Then $f(26) = \{2, 13\}$, $g(26) = \{7, 11, 13, 17, 19, 23\}$,

$$(f \cup g)(26) = \{2, 13\} \cup \{7, 11, 13, 17, 19, 23\} = \{2, 7, 11, 13, 17, 19, 23\},$$

$$(f \cap g)(26) = \{2,13\} \cap \{7,11,13,17,19,23\} = \{13\},$$
$$(f - g)(26) = \{2,13\} - \{7,11,13,17,19,23\} = \{2\},$$
$$(f \oplus g)(26) = \{2,13\} \oplus \{7,11,13,17,19,23\} = \{2,7,11,17,19,23\},$$

and $\overline{f}(26)$ is the set of all prime numbers, with the exceptions of 2 and 13. □

1.4.5 Functions of several variables

So far we have focused entirely on functions of a single variable, but most functions in practice involve functions that depend on several variables. For example, a student's final grade in a course is often a (linear) function of several different assessments of competency, including those for written exams, projects, and research papers. Furthermore, in the era of "big data" it is not uncommon to define functions based on hundreds or even thousands of variables, where the rules of such functions are obtained via machine-learning algorithms. Some practical uses of such functions include recognizing sounds and images, computing credit scores, and detecting dangerous events, such as fraud, disease, and breaches of security.

Definition 1.4.3. Given n variables $x_1, \ldots, x_n$, where $A_i = \text{dom}(x_i)$, $i = 1, \ldots, n$, A **function of n-variables** over $x_1, \ldots, x_n$, denoted

$$f(x_1, \ldots, x_n),$$

and having codomain B, is a function $f : A_1 \times \cdots \times A_n \to B$ that assigns every n-tuple $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$ to some member $b \in B$. In function notation this is written as $f(a_1, \ldots, a_n) = b$.

Thus, a function of n variables is really just a function of a single variable, where the domain of that single variable is the set of all n-tuples of

$$A_1 \times \cdots \times A_n.$$

Example 1.4.14. The volume $v(r, h)$ of a cylinder is a function of two variables, namely the radius r of its circular cross section, and its height h, where the domain set of both r and h is $\mathcal{R}^+$, the set of positive real numbers. In this case the rule for $v(r, h)$ is $v(r, h) = \pi r^2 h$. For example, $v(2, 3) = \pi 2^2 \cdot 3 = 12\pi$.

Although, technically speaking, a function f of n variables has a single domain set, namely $A_1 \times \cdots \times A_n$, we nevertheless refer to the sets $A_1, \ldots, A_n$ as the domain ses of f. Moreover, the **signature** of f is denoted $f(A_1, \ldots, A_n, B)$. For example, the signature of the volume function $v(r, h)$ in the previous example is $v(\mathcal{R}^+, \mathcal{R}^+, \mathcal{R}^+)$, where the first two $\mathcal{R}^+$'s represent the domains of r and h, while the third represents v's codomain.

An abridged version of a function's signature may be obtained by replacing sequences of repeating sets with a single occurrence of the set, followed by the number of times the set appears in the sequence. For example, the signature of $v(r, h)$ may be reduced to $v(\mathcal{R}^+, 3)$, since $\mathcal{R}^+$ appears three consecutive times in the unabridged signature.

Just as a function of a single variable is called a **unary function**, so a function of two variables is called a **binary function**. Moreover, every binary arithmetic, set, or logical operation may be considered a binary function. For example, binary addition has the signature $+(\mathcal{R}, 3)$, while binary OR has the signature $\vee(\mathcal{B}, 3)$. Although it is perfectly legal to write $+(3, 1) = 4$, we usually prefer to use $+$ with *infix* notation, where the first input is written before $+$, and the second input is written after; such as $3 + 1 = 4$. Section 1.6 elaborates more on the ways that one can position a function's name in relation to its inputs.

1.5 Predicate logic

Section 1.3 introduced several Boolean operations, including AND, OR, and NOT. Each is an example of a **Boolean function**, i.e. a function whose domain and codomain sets are all Boolean. On the other hand, if we allow for a function of one or more variables to have domain sets of any type, yet still require a Boolean codomain, then we have what is called a **predicate function**. Predicate functions are often used to determine whether or not a member of some set has a given property. For example, let n be a positive integer, and consider the predicate function $p(n)$ that assigns n the value `true` iff n is a prime number. Then for any positive integer n we may evaluate $p(n)$ to determine if n has the property of being prime.

Example 1.5.1. Consider the set membership operation $\in$. We may view this as a binary predicate function whose first input is a member u of some universe $\mathcal{U}$, whose second input is a subset A of $\mathcal{U}$, and whose output is `true` iff $u \in A$. For example, if $\mathcal{U} = \mathcal{N}$, then $3 \in \emptyset$ evaluates to `false`, while $5 \in \{2, 3, 5, 7\}$ evaluates to `true`. In general, $\in$ has the signature $\in (\mathcal{U}, \mathcal{P}(\mathcal{U}), \mathcal{B})$. $\square$

Given a predicate function $f : A \to \mathcal{B}$, in order to provide a rule for f, we must have access to operations that take members from A and assign to them a Boolean value. Although most of these operations will depend on the type of domain that A represents, there are two binary operations that are universal to all sets, namely $=$ and $\neq$. Indeed, if x and y are variables having domain A, then $x = y$ evaluates to `true` iff x and y have been assigned the same domain member of A. On the other hand $x \neq y$ evaluates to `true` iff x and y have been assigned different domain members. Thus, evaluating the operations $=$ and $\neq$ amounts to being able to distinguish members of A.

Example 1.5.2. Let A denote the power set of $\{1, 2, 3\}$. Then $=$ and

$\neq$ are two predicate functions defined on A, For example, $\{1, 2\} = \{1, 3\}$ evaluates to `false`, while $\{1, 2\} \neq \{1, 3\}$ evaluates to `true`. $\square$

1.5.1 Relational operations

The above two equality operations are examples of what is called a **relational operation**. As the name suggests, a relational operation is a binary operation that tests whether or not two domain members are related in some way. For example, `equals` tests if two members are identical. Although most other relational operations apply only to a particular domain, the relational operations $<$, $\leq$, $>$, and $\geq$ apply to any domain whose members are ordered in some way. Of course, the most common ordered domains are the natural, integer, and real numbers. Indeed, statements like $3 < 5$ and $-2.754 \geq -4.13$ are quite common in these domains. For this reason each of these operations is called an **ordering relation**. Generally speaking, any finite or countably infinite set can be ordered so as to allow the use of ordering relations.

Example 1.5.3. Let $A = \{a, b, c, d, \ldots, z\}$ denote the set of alphabet letters. This set has a familiar ordering taught in elementary school, and that statements like $t < v$ and $s \geq s$ true, and statements like $c > z$ and $m \leq b$ false. $\square$

Example 1.5.4. The binary set operation $\subseteq$ is an example of a relational operation over any collection of sets $\mathcal{S}$ since, for any $A, B \in \mathcal{S}$, $A \subseteq B$ evaluates to either `true` or `false`. In this case $\subseteq$ has the signature $\subseteq (\mathcal{S}, 2, \mathcal{B})$. $\square$

1.5.2 Boolean operations on predicate functions

Recall from Section 1.4.4 that if $f, g : A \to B$ are functions, and B is closed under some binary operation, then we may apply this binary operation to functions f and g. In particular, if f and g are predicate functions, then the binary operations AND, OR, NOT, etc. may be applied to f and g.

Definition 1.5.1. Let $f, g : A \to \mathcal{B}$ be predicate functions, Then the following functions are well-defined for all $a \in A$.

Not $(\neg f)(a) = \neg f(a)$

And $(f \wedge g)(a) = f(a) \wedge g(a)$

Or $(f \vee g)(a) = f(a) \vee g(a)$

Exor $(f \oplus g)(a) = f(a) \oplus g(a)$

IfThen $(f \to g)(a) = f(a) \to g(a)$

Equivalence $(f \leftrightarrow g)(a) = f(a) \leftrightarrow g(a)$

Example 1.5.5. Suppose we have a definition for the relational operations $x < y$ and $x = y$. Then the remaining operations $x \neq y$, $x \leq y$, $x > y$, and $x \geq y$ may all be defined by using $x < y$, $x = y$, and the Boolean operations on predicate functions. Indeed, for all x and y, we have

$$(x \neq y) = \neg(x = y),$$
$$(x \leq y) = ((x < y) \vee (x = y)),$$
$$(x > y) = \neg(x \leq y) = (\neg((x < y) \vee (x = y))),$$

and

$$(x \geq y) = ((x > y) \vee (x = y)) = ((\neg((x < y) \vee (x = y))) \vee (x = y)) =$$
$$\neg(x < y) \vee (x = y).$$

□

Quantifiers

The predicate functions together with the Boolean operations on predicate functions constitute two of the three fundamental building blocks for constructing a statement in **predicate logic**. In this section we study the third and final ingredient: quantifiers. To motivate the need for quantifiers, recall that logical statements provide a formal means for representing truth about a given system or theory. Moreover, such a statement should i) be able to be evaluated as either being `true` or `false`, and ii) be as succinct as possible so as not to obfuscate its meaning. Now consider the predicate function $f : P \to \mathcal{B}$, where P is a set of people, and $f(p)$ asserts that person p had fun at the amusement park. Furthermore, suppose we want to use f to make the statement "everyone had fun at the amusement park". As a first attempt, suppose we model this statement as "$f(p)$", where $\text{dom}(p) = P$. The problem with this is that this statement can only evaluate to `true` or `false` when p has been assigned a specific person. For example, if John had fun at the amusement park, then $f(\text{John}) = \texttt{true}$. Thus requirement i) is not met. As a second attempt, we could make a conjunctive statement that uses every member in P. For example, suppose $P = \{p_1, \ldots, p_n\}$ consists of n people, then to express that everyone had fun at the amusement park, we could write

$$f(p_1) \wedge f(p_2) \wedge \cdots \wedge f(p_n).$$

This statement is well-constructed, and evaluates to `true` iff everyone had fun, and so has the correct meaning. However, this statement may not be succinct for large values of n. Furthermore, we may want to make the statement about everyone having fun, but perhaps prefer to keep the attendee's names private. The **universal quantifier** $\forall$ solves both these issues. Indeed, the statement

$$\forall p f(p)$$

reads as "for all members p of P, $f(p)$ is true".

The universal quantifier $\forall$ is an example of what is called an **iterator function** because it works in iterative steps, where each iteration involves assigning a variable v, called the **index variable**, one of its domain values, followed by evaluating an expression that depends on v, and combining the answer with the answers obtained in previous iterations. An iterator only stops until the desired final answer has been computed, which may take at most n iterations, where $n = |\text{dom}(v)|$. For example, in the statement $\forall p f(p)$, p serves as the index variable, while $f(p)$ is the expression that gets repeatedly evaluated, once for each member of p's domain. In this case the $\forall$ iterator is looking for a person p in domain set P for which $f(p) = 0$. If such a person is found, then the final answer is `false`, since it is not true that everyone had fun at the amusement park. On the other hand, after iterating through all the members of P and discovering that $f(p) = 1$ for each member, then the final answer is `true`, since everyone had fun.

Example 1.5.6. Let variable n have domain $\{1, \ldots, 10\}$, and let prime(n) denote the predicate function that evaluates to `true` iff n is prime. Then the statement $\forall n \text{prime}(n)$ evaluates to `false` since are members (e.g. 1,4, 6, 8, 9,10) of n's domain that are not prime numbers. Also, $\forall n \neg \text{prime}(n)$ also evaluates to `false` since it is saying that every member is not prime, which is also not true, since 2,3,5, and 7 are prime. □

Another iterator that is perhaps more familiar in mathematics is the $\sum$ iterator. Similar to $\forall$, $\sum$ has two inputs: an index variable v and an expression. However, whereas $\forall$ requires a Boolean expression (i.e. one that evaluates to a Boolean value), $\sum$ requires a numerical expression that is repeatedly evaluated using different members of v's domain, and all the answers are summed together to form the final answer. For example, let i be an index variable having domain $\{1, 2, 3, 4, 5\}$. Then, to evaluate

$$\sum_{i=1}^{5} 3i,$$

we must evaluate $3i$ for each domain value of i, and sum the results. This procedure yields

$$\sum_i 3i = \sum_{i=1}^{5} 3i = 3(1) + 3(2) + 3(3) + 3(4) + 3(5) = 45.$$

The convention of placing $i = 1$ under $\sum$, and 5 above $\sum$ is called **summation notation**, and is usually used when the variable domain is a contiguous integer sequence of the form $\{a, a+1, a+2, \ldots, b\}$. In this case we would have

$$\sum_{i=a}^{b}.$$

Example 1.5.7. Consider the case when variable j has domain $\{0.1, 0.4, 0.5\}$. In this case the domain is not a contiguous integer sequence, but we may still use the notation

$$\sum_j (j - 2j^2),$$

since j's domain is understood. In this case we get

$$\sum_j (j - 2j^2) = 0.1 - 2(0.1)^2 + 0.4 - 2(0.4)^2 + 0.5 - 2(0.5)^2 = 0.16$$

□

Example 1.5.8. Another mathematical iterator that one may have encountered before is the **product iterator** $\prod$ which is similar to the sum iterator, with the exception that now the answers are multiplied instead of added. For example, if i has domain 1 through 10, then

$$\prod_i i = 1 \cdot 2 \cdot 3 \cdots \cdot 10 = 10! = 3628800.$$

□

Returning to the quantifier iterators used in predicate logic, the **existential quantifier**, $\exists$, works in the same manner as the universal quantifer, but now the statement $\exists p f(p)$ reads “there exists a p for which $f(p)$ is true”. For example, letting f denote the “had fun at the amusement park” predicate, in layperson terms $\exists p f(p)$ states that at least one person (may be more) had fun at the amusement park. Moreover, if the domain set of p is $P = \{p_1, \ldots, p_n\}$, then $\exists p f(p)$ has the same logical meaning as

$$f(p_1) \vee f(p_2) \vee \cdots \vee f(p_n).$$

Thus, $\exists$ represents the dual of $\forall$, meaning that, for any tautology there is another tautology that can be formed by replacing $\forall$ with $\exists$ (and vice versa), AND with OR, and 1 with 0.

Example 1.5.9. Consider the two predicate-logic statments $\forall x(p(x) \wedge q(x))$ and $\forall x p(x) \wedge \forall x q(x)$. These statements are logically equivalent, since their evaluations will always result in the same truth value. To see this, suppose the domain of x is a group of people, $p(x)$ stands for “x likes pizza” and $q(x)$ stands for “x likes quiche”. Then the first statement is saying “everyone likes both pizza and quiche”, while the second is saying “everyone likes pizza and everyone likes quiche”. Both are two ways of saying that everyone in the domain likes both these kinds of foods. Therefore we have the identity

$$\forall x(p(x) \wedge q(x)) = \forall x p(x) \wedge \forall x q(x),$$

whose corresponding dual identity is

$$\exists x(p(x) \vee q(x)) = \exists x p(x) \vee \exists x q(x).$$

□

Example 1.5.10. We now show that the predicate-logic statments

$$\forall x(p(x) \vee q(x))$$

and $\forall x p(x) \vee \forall x q(x)$ are *not* logically equivalent. To do this we must provide an example of two predicate functions p and q, and a domain for x for which the two statements have different truth evaluations. Using the same predicate functions p and q from the previous example, suppose the domain of x is $\{\text{Alice}, \text{Bob}\}$. Moreover, suppose Alice likes pizza but not quiche, and Bob likes quiche but not pizza. Then $\forall x(p(x) \vee q(x))$ evaluates to `true` since both Alice and Bob like either pizza or quiche (Alice likes pizza, Bob likes quiche). However, $\forall x p(x) \vee \forall x q(x)$ evaluates to `false` since neither everyone likes pizza (Bob does not like it) nor everyone one like quiche (Alice does not like it). We leave it as an exercise to show that

$$\exists x(p(x) \wedge q(x))$$

and $\exists x p(x) \wedge \exists x q(x)$ are *not* logically equivalent. □

The previous two examples address the ability (and inability) for the universal and existential quantifiers to distribute over a binary operation (such as AND and OR). The reader is encouraged to continue this study using the other logic operations, such as IF-THEN, EXOR, and EQUIVALENCE.

It's important to note that the truth or falsehood of a predicate-logic statement will often depend on the choice of domains for each of it's index variables. For example, the statement $\exists x(x < 0)$ is true if $\text{dom}(x) = \mathcal{I}$, but is false if $\text{dom}(x) = \mathcal{N}$. Also, the choice of domain can often affect the structure of the statement itself.

Example 1.5.11. Let $p(x)$ be the predicate function that evaluates to `true` iff person x has programmed with the Python programming language, where the domain of x is the set of all people with computer science degrees. Let $\text{cs}(y)$ be the predicate function that evaluates to `true` iff person y has a computer science degree, where the domain of y is the set of all living people. Now consider the statement "Every person with a CS degree has programmed with Python." Using variable x, this statement may be modeled as

$$\forall x p(x). \tag{1.1}$$

However, using variable y, $\forall y p(y)$ implies that every living person has programmed with Python, which is too strong a statement. To weaken the statement we must place a condition on when someone has programmed with Python, namely the condition that the person has a CS degree. In other words, for any living person, if that person has a CS degree, then they have programmed with Python". This is modeled as

$$\forall y(\mathrm{cs}(y) \to p(y)). \tag{1.2}$$

To see that this statement has the same truth value as 1.1, first assume that the former evaluates to `true`. Now consider an arbitrary living person y. If this person has no CS degree, then $\mathrm{cs}(y)$ is false which makes the statement $\mathrm{cs}(y) \to p(y)$ vacuously true. On the other hand, if y does have a CS degree, then, since $\mathrm{cs}(y)$ is true, y is a member of the domain of x, in which case $p(y)$ is true. Hence, $\mathrm{cs}(y) \to p(y)$ evaluates to (`true` $\to$ `true`) $=$ `true`. We leave it as an exercise to show that the truth of 1.2 implies the truth of 1.1. $\square$

The quantifiers introduced thus far allow us to state that all members of some domain possess a property, or that at least one member does. But what if we want to say that at least one half of the members possess a property, or that no more than 20 possess the property? The following extensions of the existential quantifier allow us to make such statements.

$\exists! x p(x)$ There exists a unique domain member a of x for which $p(a)$ evaluates to `true`.

$\exists^{=n} x p(x)$ There exist exactly n domain members a of x for which $p(a)$ evaluates to `true`.

$\exists^{<n} x p(x)$ There exist fewer than n domain members a of x for which $p(a)$ evaluates to `true`.

$\exists^{>n} x p(x)$ There exist more than n domain members a of x for which $p(a)$ evaluates to `true`.

$\exists^{\leq n} x p(x)$ There exist no more than n domain members a of x for which $p(a)$ evaluates to `true`.

$\exists^{\geq n} x p(x)$ There exist at least n domain members a of x for which $p(a)$ evaluates to `true`.

Example 1.5.12. Suppose $f(p)$ is the predicate function that evaluates to `true` iff person p had fun at the amusement park, where the domain of p is a set of 1,000 people. Then the statement "anywhere between 80 to 90 percent of the people had fun at the amusement park" can be modeled with the statement

$$\exists^{\geq 800} p f(p) \wedge \exists^{\leq 900} p f(p).$$

□

Before leaving this section it is worth to note in passing that there are De Morgan-like identities for quantifier statements, namely

$$\neg \forall x p(x) = \exists x \neg p(x) \tag{1.3}$$

and

$$\neg \exists x p(x) = \forall x \neg p(x). \tag{1.4}$$

For example, it's not true that everyone likes pizza if and only if it is true that at least one person does not like pizza. Similarly, it's not true that someone likes pizza if and only if it is true everyone does not like pizza.

1.5.3 Set-builder notation

The list notation introduced in Section 1.1 represents the most direct way to describe the members of a set S. However, this notation has its limitations, especially for the case when a set has very large cardinality, and there is no starting sequence of members (such as $\{1, 2, 3, \ldots\}$) that captures the entire set. On the other hand, suppose we have a predicate function $p(x)$, where i) $S \subseteq \text{dom}(x)$, and ii) $p(x) = 1$ iff $x \in S$. Then **set-builder notation** allows us to describe the set as

$$\{x|p(x)\},$$

which reads "the set of domain members a of x for which $p(a)$ evaluates to `true`". When using set-builder notation, we must specify the domain of x in case there is a chance that it may not be understood.

Example 1.5.13. Let $\text{div}(x, y) = 1$ iff integer x divides evenly into integer y. Then we may use div and set-builder notation to describe the set of integers between 1 and 1000 that are divisible by either 2 or 3. Indeed, letting x have domain equal to $\mathcal{I}$, this set is equal to

$$\{x|x \geq 1 \wedge x \leq 1000 \wedge (\text{div}(2, x) \vee \text{div}(3, x))\}.$$

Note that, had we stated the domain of x as all integers between 1 and 1000, then the above simplifies to

$$\{x|\text{div}(2, x) \vee \text{div}(3, x)\},$$

since the domain of x is already understood. □

1.5.4 Quantifying predicates of several variables

In the previous section we studied the use of quantifiers for a predicate-logic statement that has a single variable. In section we extend the study to statments involving two or more variables.

To begin, consider two disjoint sets of people A and B, where the members of A and B live on Alameda Island and Balboa Island, respectively. Next, let predicate function $c(x, y)$ evaluate to `true` iff person x has ever phoned person y, where x and y have domains A and B, respectively. Since there are two variables, there are 8 different ways to quantify $c(x, y)$. For example, $\exists x\forall y c(x, y)$, $\forall y\exists x c(x, y)$, and $\exists x\exists y c(x, y)$ are three such ways. In what follows we provide a definition for how to interpret the meaning of each one, and interpret this meaning using $c(x, y)$.

$\exists x\exists y c(x, y)$ There is some member $a \in A$, and some member $b \in B$ for which $c(a, b) = 1$. Example: someone on Alameda (and belonging to set A) has phoned someone on Balboa (and belonging to set B).

$\exists y\exists x c(x, y)$ There is some member $b \in B$, and some member $a \in A$ for which $c(a, b) = 1$. Example: someone on Balboa has received a call from someone on Alameda.

$\exists x\forall y c(x, y)$ There is some member $a \in A$ such that, for every member $b \in B$, $c(a, b) = 1$. Example: someone on Alameda has phoned everyone on Balboa.

$\exists y\forall x c(x, y)$ There is some member $b \in B$ such that, for every member $a \in A$, $c(a, b) = 1$. Example: someone on Balboa has received a call from everyone on Alameda.

$\forall x\exists y c(x, y)$ For every member $a \in A$ there is a member $b \in B$ for which $c(a, b) = 1$. Example: everyone on Alameda has phoned someone on Balboa.

$\forall y\exists x c(x, y)$ For every member $b \in B$ there is a member $a \in A$ for which $c(a, b) = 1$. Example: everyone on Balboa has received a call from someone on Alameda.

$\forall x\forall y c(x, y)$ For every member $a \in A$ and every member $b \in B$, $c(a, b) = 1$. Example: everyone on Alameda has phoned everyone on Balboa.

$\forall y \forall x c(x,y)$ For every member $b \in B$ and every member $a \in A$, $c(a,b) = 1$. Example: everyone on Balboa has received a call from everyone on Alameda.

One way to visualize each of the above statements is to draw a **binary predicate graph** whose left and right nodes represent members of A and B, respectively. Furthermore, a line segment joins a left node labeled with $a \in A$ with a right node labeled with $b \in B$ iff $c(a,b) = 1$. Figure 1.5.1 shows a binary predicate graph for the case when $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $B = \{b_1, b_2, b_3, b_4, b_5\}$. Notice that, for the statement $\exists x \forall y c(x,y)$ to be true, there needs to be at least one member of A that is joined with every member of B. Looking at the graph, we see that there are actually two such members: a_1 and a_3. Therefore, $\exists x \forall y c(x,y)$ is a true statement.

As another example, Figure 1.5.2 shows another binary predicate graph that defines a predicate $c(x,y)$ for which $\forall y \exists x c(x,y)$ evaluates to `true`. To see this, notice that every member of B is joined with a least one member of A.

Example 1.5.14. Suppose predicate function depart(f,a) evaluates to `true` iff flight f departs from airport a, and carrier(f,c) evaluates to `true` iff flight f is owned by carrier c. The problem is to use these predicate functions to model the statement "every carrier owns a flight that departs from Los Angelest International Airport (LAX)". Translating a natural-language statement into predicate logic is often an iterative process, where we write an answer, analyze it for correctness, and make changes where necessary. In this case we are making a statement about *every* carrier, namely that it owns a flight that departs from LAX. So we might start off and write

$$\forall c(\text{carrier}(f,c) \wedge \text{depart}(f,\text{LAX})).$$

Here we use $\wedge$ since there are two things that are both true about flight f, namely that it's owned by c and departs LAX. This is almost what we want,

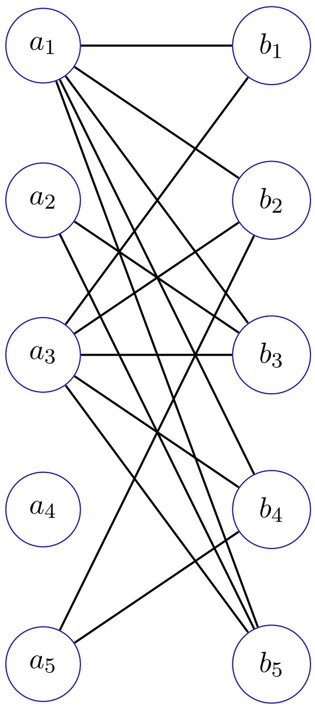

Figure 1.5.1: A binary predicate graph that defines $c(x, y)$ so that $\exists x \forall y c(x, y)$ evaluates to `true`

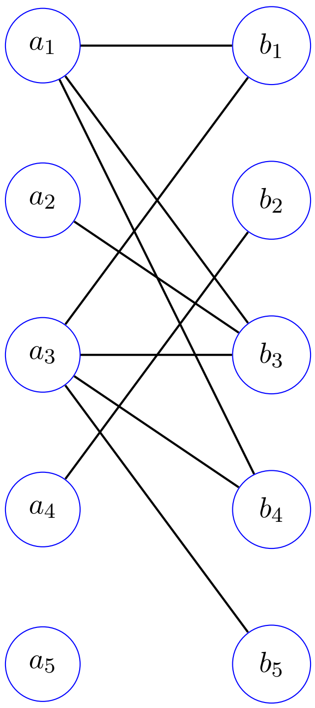

Figure 1.5.2: A binary predicate graph that defines $c(x, y)$ so that $\forall y \exists x c(x, y)$ evaluates to `true`

but our answer treats f as a **free variable**, meaning that it is not indexed by some quantifier. Moreover, when a variable is free, it means that the statement cannot be evaluated until f has been assigned a particular flight in its domain. But the problem does not mention any particular flights that pertain to f's domain. Thus, it seems more appropriate to index f with a quantifier. So as a second attempt we may write

$$\forall c \forall f(\text{carrier}(f, c) \wedge \text{depart}(f, \text{LAX})).$$

Staring at this answer, we notice that it seems too strong, in that it is saying that, for any carrier c, and for any flight f, the carrier owns f and f departs LAX. This means that every carrier owns every flight, which seems nonsensical, since a flight is usually owned by a single carrier. Finally, if we change $\forall f$ to $\exists f$, then we arrive at

$$\forall c \exists f(\text{carrier}(f, c) \wedge \text{depart}(f, \text{LAX})),$$

which now seems consistent with the original statement. Now, for any carrier c, we are just asking for one flight f that is both owned by c and departs LAX. □

The following are some guidelines for modeling a natural-language statement with a predicate logic formula.

1. **Identify all relevant domains.** Modeling a statement with predicate logic is typically done in relation to modeling some problem of interest, which itself has a set of relevant domains. For example, if the problem is to develop a schedule of courses to offer at a university, then the relevant domains may include course subjects, rooms, instructors, students, and times. Then a statement such as "an instructor cannot teach two different courses at the same time", refers to the instructor, course, and time domains.

2. **Identify all constants.** Constants arise when a statment makes reference to a particular member of some domain. For example, the statement "Dr. Gold is only able to teach courses that start at or after 5:00 pm" makes reference to two constants: Dr. Gold and 5:00 pm. These constants will appear as inputs to one or more predicate functions, such as $\text{start}(c, \text{5:00 pm})$ and $\text{ins}(c, \text{Gold})$, where the domain of c is the set of all courses.

3. **Identify all variables.** The use of variables arises when a statement makes references to generic members of some domain. For example, the statement "an instructor cannot teach two different courses at the same time" makes generic references to instructors, courses, and times. Thus, a model of this statement should make use of variables, such as i, c, and t, whose domains correspond with the domains mentioned in the statement.

4. **Identify the predicate functions needed to support the statement.** Since the statement being made is asserting a truth about the problem being modeled, there must be one or more predicate functions that together (with the help of logical operations) help formally represent this truth. For example, the statement "an instructor cannot teach two different courees at the same time", we can support this statement using three predicate functions $\text{teach}(c, i)$, which is true iff instructor i teaches course c, $\text{time}(c, t)$ which is true iff course c is taught during time (interval) t, and $\text{overlap}(t_1, t_2)$ which is true iff time intervals t_1 and t_2 overlap.

5. **Provide an initial predicate-logic formula without quantifiers.** This step allows one to provide the basic logical structure of the statement without getting bogged down by the types of quantifiers needed for each variable, and the ordering of the quantifiers. Using our running example, we want to imagine an instructor who is teaching two or more courses, and then assert that the time intervals of these courses do not overlap. Since we are thinking of two different courses, it seems appropriate to have two course variables c_1 and c_2.

Moreover, we do not want these courses to overlap in time *provided* the instructor is teaching both of them. This suggests the if-then statement

$$(\text{teach}(c_1, i) \wedge \text{teach}(c_2, i) \wedge \text{time}(c_1, t_1) \wedge \text{time}(c_2, t_2)) \rightarrow \neg\text{overlap}(t_1, t_2).$$

6. **Assign quantifiers to each variable and assign an order to the quantifiers.** The predicate-logic formula provided in the previous step cannot evaluate to `true` or `false` until all variables are indexed by a quantifier. A universal quantifier is appropriate when the statement is referring to all generic members. For example, we know that *all* instructors have a real problem when it comes to teaching two different courses at the same time. On the other hand, an existential quantifier seems appropriate when the generic member represents some particular member, but not too particular to where it where it can be identified with a constant. In other words, existential is appropriate for a generic member that possesses some particular property that is not necessarily possessed by every member. For example, recall the statement "every carrier has some flight that departs LAX". This statement suggests two variables c and f, with c being assigned a universal quantifier, and f being assigned an existential one. This is because the statement refers to all carriers, and to a particular (albeit generic) flight owned by that carrier. Finally, the order of $\forall c \exists f$ means "for every carrier this a flight owned by that carrier", where as $\exists f \forall c$ means "there is a flight such that every carrier owns that flight" which is not what we want. Indeed, to verify the statement we would first go through a list of all carriers, and, for each carrier in the list, we would have to examine all the flights owned by the carrier, looking for one that departs LAX. On the other hand, to verify the statement "there is a course for which every student received an A grade", we would first look through a list of courses, searching for at least one course whose gradebook lists final grades consisting of all A's.

   Returning to our running example, it seems appropriate to assign a universal quantifier to each of the variables i, c_1, c_2, t_1, and t_2. This is because the statement applies to all instructors, courses, and times. In other words, if you are given any instructor, any two courses, and any two times, and are told that the instructor is teaching the two courses, and that the two times correspond with the times of the courses, then you must conclude that the two times do not overlap. Therefore we have the statement

$$\forall i \forall c_1 \forall c_2 \forall t_1 \forall t_2 (\text{teach}(c_1, i) \wedge \text{teach}(c_2, i) \wedge \text{time}(c_1, t_1) \wedge \text{time}(c_2, t_2)) \\ \rightarrow \neg\text{overlap}(t_1, t_2). \tag{1.5}$$

Notice that, since the quantifiers are all universal, the order in which they are written does not matter.

7. **Check for exceptional cases that may invalidate the formula.** At times a formula that seems correct will actually fail for one or more exceptional cases. And usually such exceptions are of a technical nature, rather than involving the actual problem being modeled. For example, in formula 1.5 above, since c_1 and c_2 have the same domain set of all courses, there will be times when c_1 and c_2 simultaneously assume the same course, in which case the times t_1 and t_2 should coincide as well, rather than not overlap. Thus, we need the added conditional hypothesis that $c_1 \neq c_2$, which gives the revised formula

$$\forall i \forall c_1 \forall c_2 \forall t_1 \forall t_2 (\text{teach}(c_1, i) \wedge \text{teach}(c_2, i) \wedge \text{time}(c_1, t_1) \wedge \text{time}(c_2, t_2) \wedge \\ c_1 \neq c_2) \rightarrow \neg\text{overlap}(t_1, t_2).$$

Example 1.5.15. Let's see how the above guidelines help to model the statement "with the exception of zero, every natural number is preceded by some other natural number".

1. The statement is about the domain of natural numbers.
2. The number 0 is the only particular natural number mentioned in the statement.

3. The statement speaks of two generic numbers, where the first number, call it x, is some arbitrary natural number, while the second number, call it y, is a natural number that precedes it.

4. To say that y comes before x requires the binary relation $<$, while $\neq$ is needed to assert that $x \neq 0$.

5. A plausible first attempt at a predicate logic formula is

$$x \neq 0 \wedge y < x.$$

6. The statement implies that x can be any natural number (except 0) while y is a particular number that depends on x. For example, if $x = 7$, then y should be a number less than 7, such as 6. Therefore, the quantifier sequence $\forall x \exists y$ seems appropriate.

7. The formula is now

$$\forall x \exists y (x \neq 0 \wedge y < x).$$

Unfortunately, this does not work for $x = 0$, since $0 \neq 0 \wedge y < 0$ is a false statement since $0 \neq 0$ is false. Rather than using $x \neq 0$ as a statement that must be true, instead it may serve as a condition that must be true in order for $y < x$ to be true for some y. Thus we have

$$\forall x \exists y (x \neq 0 \rightarrow y < x).$$

Equivalently, since y does not appear in the hypothesis, $\exists y$ may be moved inward to attain the statement

$$\forall x (x \neq 0 \rightarrow \exists y (y < x)).$$

□

Example 1.5.16. We translate the statement "between any two real numbers there is another real number that lies between them" into predicate logic. This statement makes reference to the domain of real numbers.

It does not mention any particular real number, but does mention three generic real numbers, call them x, y, and z, where x and y are the "any two numbers", and z is the third number that "lies between them". Using the less-than relation, we may form the initial quantifier-free predicate formula

$$x < z \wedge z < y.$$

Adding quantifiers, we assign universals to x and y, since x and y are any two real number, and assign an existential to z, since z is depends on x and y (i.e. choosing any z will not always work). This yields

$$\forall x \forall y \exists z (x < z \wedge z < y).$$

Finally, this formula works for almost all cases, except for when $x = y$. So we must add the condition $x \neq y$ to obtain the final formula

$$\forall x \forall y (x \neq y \rightarrow \exists z (x < z \wedge z < y)).$$

$\square$

1.6 Expressions

Informally, an **expression** is a formula that is comprised of one or more functions, and is capable of being evaluated to obtain some output. Perhaps the most common type of expression is the **arithmetical expression**, which is comprised of parentheses, arithmetic operations, and numbers. For example, $(5 + 8) + 3 \times 7$ is an arithmetic expression. Such expressions represent ways of combining arithmetic operations in order to perform more complex computations. This idea generalizes to expressions that are comprised of functions of any signature-type.

Before defining functional expressions, recall that an arithmetic expression may be evaluated to some final number. For example, $(5 + 8) + 3 \times 7$

evaluates to $(13) + 3 \times 7 = 13 + 21 = 34$. In general, if e is an expression (arithmetic or otherwise), eval(e) denotes the final value one gets when evaluating e. For example,

$$\text{eval}((5 + 8) + 3 \times 7) = 34.$$

Before providing a more formal definition of an expression, it's worth emphasizing that an expression has both *meaning* and *syntax*. For example, the expression $(5+8)+3\times 7$ evokes the meaning of adding 5 to 8, followed by adding the sum to the product of 3 and 7. It is how we interpret the expression. On the other hand, by "syntax" we mean that it is also a string of symbols that was formed by obeying certain rules. Some of the rules are as follows.

1. A left parenthesis must be paired with a right parenthesis to encapsulate a valid subformula.
2. Two numbers (or subformulas) being added must be respectively placed to the left and right of the $+$ symbol.
3. Two numbers (or subformulas) being multiplied must be respectively placed to the left and right of the $\times$ symbol.

1.6.1 Function notations

Since expressions are comprised of functions and have a syntactical aspect, we should first examine the four different kinds of syntax used for writing a function and its domain inputs They are four notations known as prefix, infix, postfix, and delimiter notation.

To begin, **prefix notation** is the most commonly used notation in both mathematics and computer programming, and has the form $f(i_1, i_2, \ldots, i_n)$, where f is the function name, and $i_1, \ldots, i_n$ are strings that represent the function's inputs. For example, consider the function named `area` that takes two real inputs l and w and outputs the product lw; i.e. the area of a rectangle having length l and width w. Then the statements $\text{area}(3.4, 3.8)$, $\text{area}(l, 7.99)$, and $\text{area}(l, w)$ are all examples of using prefix notation. Prefix notation is so common that it is often referred to as simply *function notation.*

The parentheses used in prefix notation are often omitted in the case where f is unary and has a symbolic name. For example, the expression $\neg(p)$, may be written without parentheses as $\neg p$, and the same goes with $-(x) = -x$. In a later section we discuss rules for eliminating parentheses for functions having two or more inputs.

Recall that **infix notation** is used for binary operations, such as $+$, $\times$, $\wedge$, and $\vee$. The general form is i_1 op i_2, where op is the operation name/symbol, and i_1 and i_2 are strings representing the two inputs. For example, $5 \times z$ and $p \vee$ `true` both use infix notation.

Next, **postfix notation** is used for some unary functions, and is similar to prefix notation, but with the function name coming after the input. For example, let $\uparrow$ be a postfix operation for which $x \uparrow$ denotes x^x. Then a statement such as $3 \uparrow +5$ evaluates to $3^3 + 5 = 27 + 5 = 32$. In a later section we discuss an important programming use of postfix notation that involves acessing the attribute values of a data structure.

Finally, **delimiter notation** is used for functions that define collections of things, such as sets and tuples. For example, we may view the set $\{1, 5, 6\}$ as a functional expression, for which the inputs, 1, 5, and 6, are delimited by a pair of braces that represents an application of the **set-constructor** function which outputs a set whose members are 1, 5, and 6. In other words, list notation for sets may be viewed as an application of the set-constructor function { } whose inputs are the set members. In a similar manner, the tuple $(3, 3, 1, 5)$ may be viewed as an application of the **tuple-constructor** function ( ), and whose inputs are the tuple components 3, 3, 1, and 5.

1.6.2 Formally defining expressions

Functional expressions can be defined from the ground up, by i) first defining the most basic expressions, and ii) showing how already existing expressions can be used to construct new, more complex ones. Thus, an expression is an example of what is called a **recursive structure**, i.e. a structure that is composed of smaller structures of the same type. Other examples of recursive structures include sets, since the members of a sets may also be sets, and trees, since in many cases a (biological) tree branch resembles that of an entire tree.

The two most basic kinds of expressions are variables and constants. For example variables x, amount, and str, are expressions, as are constants 4, 0.3345, LAX, Julie, and `true`. Such an expression is called a **terminal**, since it cannot be reduced to a more simpler kind of structure. It's worth noting that terminals may also be considered as functions. For example, the constant 3 may be thought of as a function that always outputs the number 3. The domain of such a function is not important, since the output is always the same. Similarly, a variable x may be considered as the **identity function** $f(x) = x$. In this case the domain of f is equal to the domain of x, and $f(x)$ evaluates to whatever value has been assigned to x. For example, if $\text{dom}(x) = \{1, 2, 3\}$, then $f(1) = 1$, $f(2) = 2$, and $f(3) = 3$. In summary, terminals are the most basic kinds of expressions, and which happen to also qualify as being functions.

Example 1.6.1. Consider the variable named `amnt` whose domain is

$$\{1, \ldots, 10\}.$$

The meaning of `amnt` is that it stands for a member of dom(amnt) $= \{1, \ldots, 10\}$, while its syntax is the string "`amnt`". Similarly, the meaning of the constant `LAX`, as used in Example 1.5.14, is that of an airport in the city of Los Angeles, while its synatix is the string "`LAX`".

The next step is to show how to create new, more complex expressions from existing ones. Before doing so, we need to define what is meant by the *range* of an expression. Recall that the range of a function $f : A \to B$ is the image of A under f, i.e. the set of codomain members $b \in B$ that have a preimage a under f, meaning $f(a) = b$. Similarly, if e is an expression, then the **range** of e, denoted range(e) is the set of possible values to which e evaluates.

Example 1.6.2. Consider the arithmetical expression $6 + x \times 3$ where $\text{dom}(x) = \{1, 2, 3, 4, 5\}$. Then the evaluation of this expression depends on the value that is assigned to x. For example, if x is assigned 1, then

$$\text{eval}(6 + x \times 3) = 6 + 1 \times 3 = 9.$$

Therefore, range$(6 + x \times 3)$ is the set of values to which $6 + x \times 3$ evaluates, which is precisely

$$\{6 + 1 \times 3, 6 + 2 \times 3, 6 + 3 \times 3, 6 + 4 \times 3, 6 + 5 \times 3\} = \{9, 12, 15, 18, 21\}.$$

□

Example 1.6.3. Consider the arithmetical expression $2x+y$ where $\text{dom}(x) = \{1, 2, 3, 4\}$, and $\text{dom}(y) = \{1, 2\}$. Then there are $4 \times 2 = 8$ different ways to assign a pair of domain values, where the first value in the pair is assigned to x, and the second is assigned to y. Thus, range$(2x + y)$ is equal to

$$\{2(1) + 1, 2(2) + 1, 2(3) + 1, 2(4) + 1, 2(1) + 2, 2(2) + 2, 2(3) + 2, 2(4) + 2\} =$$

$$\{3, 5, 7, 9, 4, 6, 8, 10\}.$$

□

What is the range of a terminal expression? If terminal c is a constant, then range(c) = $\{c\}$, since a constant can only evaluate to one value. On the other hand, if terminal x is a variable, then range(x) = dom(x), since x can assume any value in its domain.

Before moving on to building more complex expressions from existing ones, it is worth noting that, if e is any expression, then so is (e). In other words, placing parentheses around an expression creates another expression, which is **evaluation-equivalent** to e, meaning both expressions will evaluate to the same value.

Now let $f : A_1 \times A_2 \times \cdots \times A_n \to B$ be a function, and suppose $e_1, \ldots, e_n$ are expressions that satisfy the property that range(e_i) $\subseteq A_i$, for all $i = 1, \ldots, n$. Then the following statements are true.

1. If f uses prefix notation, then $f(e_1, \ldots, e_n)$ is an expression.
2. If $n = 1$ and f uses prefix notation, then $f\ e_1$ is an expression.
3. If $n = 2$ and f uses infix notation, then e_1 f e_2 is an expression.
4. If $n = 1$ and f uses postfix notation, then $e_1\ f$ is an expression.
5. If f uses delimiter notation, with delimiters f_l and f_r, then

   $$f_l\ e_1, \ldots, e_n\ f_r$$

   is an expression.

Finally, in each of the above cases, the range of the expression is defined as the image of f under $\text{range}(e_1) \times \cdots \times \text{range}(e_n)$, i.e. it's the set of all codomain members $b \in B$ for which there exists a tuple $(a_1, \ldots, a_n)$, such that i) $f(a_1, \ldots, a_n) = b$, and ii) $a_i \in \text{range}(e_i)$, for all $i = 1, \ldots, n$.

Note that any expression e_1 that is used to construct an expression e_2 is called a **sub-expression** of e_2. For example, $(1 + 7)$ is a sub-expression of $(1 + 7) \times 5$, while 1 is a sub-expression of each of $1 + 7$, $(1 + 7)$, and $(1 + 7) \times 5$.

Example 1.6.4. Consider the function $\text{area}(l, w)$, where variables l and w accept nonnegative real numbers. Furthermore, suppose function $\text{meas}(a, b)$ is a function of two real variables a and b that outputs a nonnegative real number. Then the following are all valid expressions:

1. 1.6
2. -3.4
3. $\text{meas}(1.6, -3.4)$
4. 2.5
5. b
6. $\text{meas}(2.5, b)$
7. $\text{area}(\text{meas}(1.6, -3.4), \text{meas}(2.5, b))$.

For example, since $\text{meas}(1.6, -3.4)$ is a valid expression that evaluates to a nonnegative real number, the variable l can be replaced with this expression. The same is true for variable w and $\text{meas}(2.5, b)$.

In addition, suppose `area` and `meas` have the following rules.

$$\text{area}(l, w) = l \times w$$

and

$$\text{meas}(a, b) = |a - b|.$$

Then

$$|1.6 - (-3.4)| \times |2.5 - b|$$

is also a valid expression.

Finally, using the above rules, and assuming b has been assigned the value 1.5, then we may evaluate each of the expressions as follows.

1. $\text{eval}(1.6) = 1.6$
2. $\text{eval}(-3.4) = -3.4$
3. $\text{eval}(\text{meas}(1.6, -3.4)) = |1.6 - (-3.4)| = 5$
4. $\text{eval}(2.5) = 2.5$
5. $\text{eval}(b) = 1.5$
6. $\text{eval}(\text{meas}(2.5, 1.5)) = |2.5 - 1.5| = 1$
7. $\text{eval}(\text{area}(5, 1)) = 5 \times 1 = 5$

$\square$

A **single-valued** expression is one that either has no variables, or whose evaluation always results in the same value, no matter how the variables are assigned. Otherwise, the expression is said to be **multi-valued**. For example, a constant is single-valued, while a variable having a domain size of two or more is multi-valued. Also, the propositional tautology $p \to (q \to p)$ is single-valued because it always evaluates to `true`. Thus, $p \to (q \to p)$ is evaluation-equivalent to the constant expression `true`.

1.6.3 Parse Trees

A good way to visualize a functional expression is with the help of a *parse tree*. Figure 1.6.1 shows the parse tree for expression 7 from Example 1.6.4. Recall that a tree is a graph-like structure that, when drawn, consists of nodes (the circles appearing in Figure 1.6.1) and edges that are line segments connecting two nodes. When an edge connects two nodes, the higher-positioned node is called the **parent**, while the lower-positioned node is called the **child**. In other words, the upper node is the parent of the lower node, who is the child of the upper node. The node at the very top of the tree is called the **root**, and is the only node having no parent. Notice that every node is the root of its own tree, which is called a **subtree** of the entire tree. For example, the node labeled with 1.6 is itself a subtree that consists of a single node, where as the node labeled with `meas`, and whose children are labeled with 1.6 and -3.4, is a subtree consisting of three nodes. The **size** of a tree is equal to the number of nodes in the tree. Finally, a node is called a **leaf** if it has no children. Otherwise it is called **internal**.

The **parse tree** for an expression e is a tree that is constructed in a bottom-up fashion, and in a manner similar to how an expression is constructed. To construct it we use the following two rules.

1. Draw a leaf node for each terminal that appears in the expression. Label each of these nodes with the name of the terminal. Note: some nodes may be assigned the same label if a terminal appears multiple

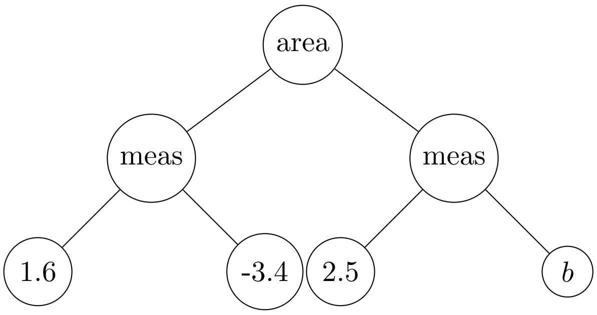

Figure 1.6.1: Parse tree for area(meas$(1.6, -3.4)$, meas$(2.5, b)$)

times in the expression.

2. Now suppose we have drawn from left to right parse trees $t_1, \dots, t_n$ for each of the respective expressions $e_1, \dots, e_n$, and $f(e_1, \dots, e_n)$ is a sub-expression of e. Then draw a new node above $t_1, \dots, t_n$, label it with f, and connect the roots of $t_1, \dots, t_n$ to this new node.

To **parse** an expression means to fully understand its structure, i.e. the set of functions that comprise the expression, along with the sub-expressions that serve as input to each function. A parse tree provides this exact information and, once constructed, allows one to easily evalute the expression based on an assignment of its variables. Evaluating an expression follows the same bottom-up method that is used to construct the parse tree. In other words, each leaf is first evaluated (easy to do since each on is either a constant or an assigned variable), follwed by evaluating any internal node whose children have already been evaluated. For example, in the case of the parse tree shown in Figure 1.6.1, and for the assigned value $b = 1.5$, the nodes may be evaluated in the following order.

1. eval$(1.6) = 1.6$

2. $\text{eval}(-3.4) = -3.4$

3. $\text{eval}(2.5) = 2.5$

4. $\text{eval}(b) = 1.5$

5. $\text{eval}(\text{meas}(1.6, -3.4)) = |1.6 - (-3.4)| = 5$

6. $\text{eval}(\text{meas}(2.5, 1.5)) = |2.5 - 1.5| = 1$

7. $\text{eval}(\text{area}(5, 1)) = 5 \times 1 = 5$

Parse trees may not seem so amazing to us humans who have learned to visually parse simple expressions, but they are essential for computers who need to process and evaluate increasingly complex expressions. For example, in later sections we describe how every statement made in the constraint-programming language that supports this book may be viewed as a functional expression, and hence every statement made in the language may be readily parsed and evaluated by following a few simple rules.

Order of operations

Parsing an expression seems mostly straightforward, but there are some subtleties that require some thought in order to get a correct parse. One such subtlety is is the order of operations in expressions that are not fully parenthesized. For example, the expression

$$(((x + 7) \times (3 - y)) \div 8)$$

is fully parenthesized (assuming parentheses around terminals is not required), and so the expression can be readily parsed, because we know that $\div$ is the final operation performed, and hence will serve as the root of the

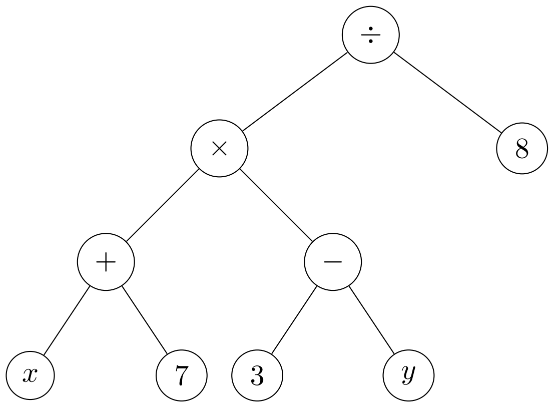

Figure 1.6.2: Parse tree for $(((x + 7) \times (3 - y)) \div 8)$

tree with 8 its right child, and a left child whose root is $\times$. The complete parse tree is shown in Figure 1.6.2

However, if all parentheses are removed then we must parse based on the order of operations: multiplication/division, followed by additon/subtraction. In this case the expression

$$x + 7 \times 3 - y \div 8$$

gets parenthesized as

$$((x + (7 \times 3)) - (y \div 8)),$$

resulting in an entirely different parse tree.

If operation op_1 is ordered before operation op_2 then the expression

$$a \ \mathrm{op}_1 \ b \ \mathrm{op}_2 \ c$$

is parsed as $(a \ \mathrm{op}_1 \ b) \ \mathrm{op}_2 \ c$. On the other hand, if op_2 comes before op_1, then it parses as $a \ \mathrm{op}_1 \ (b \ \mathrm{op}_2 \ c)$. If the operations have equal order (such as in the case of addition and subtraction) then op_1 takes precedence, since it appears first in left-to-right order. The following description provides the ordering of the operations for the arithmetic, set, relational, and Boolean operations.

Arithmetic additive inverse, multiplication/division, addition/subtraction

Set complement, intersection, union/difference, symmetric difference

Relational $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $\in$, $\subseteq$, etc. all have equal order

Boolean not, and, or, exor, if-then, equivalence

Notice that unary operations always come first in the order, and that relational operations all have equal order. Finally, arithmetic and set operations come before relational operations, which in turn come before Boolean operations. For example, taking the union of all the operations listed above and ordering them, the additive-inverse and complement operations appear first in the ordering, while logical equivalence appears last.

Example 1.6.5. Consider the expression

$$3 \times x - 5 \times y \leq 3 \div z \vee y < z \to A \oplus A \cap B - C \subseteq D.$$

Since Boolean operations are ordered last, we may begin by adding the following pairs of parentheses.

$$((3 \times x - 5 \times y \leq 3 \div z) \vee (y < z)) \to (A \oplus A \cap B - C \subseteq D).$$

Next, using the order of arithmetic operations and the fact that they appear before relational operations, we next have

$$(((3 \times x) - (5 \times y) \leq (3 \div z)) \vee (y < z)) \to (A \oplus A \cap B - C \subseteq D).$$

Finally, using the order of set operations, we arrive at the final full parenthesization

$$(((3 \times x) - (5 \times y) \leq (3 \div z)) \vee (y < z)) \to ((A \oplus ((A \cap B) - C)) \subseteq D).$$

$\square$

1.6.4 Literal expressions

A **literal expression** is any expression e whose string of symbols literally equals a member of some domain. In other words eval(e) is written exactly as e is written. For example, the constant -3789 is a literal expression since eval(-3789) $= -3789$. The expression $\{1, 3, 4, 6\}$ is also a literal because

$$\text{eval}(\{1, 3, 4, 6\}) = \{1, 3, 4, 6\}.$$

Note however that this expression not a constant, since it uses the set-constructor function { }. On the other hand, suppose variable x has been assigned the value 6. Then

$$\text{eval}(\{1, 3, x, 6\}) = \{1, 3, 6, 6\} = \{1, 3, 6\},$$

which is not the same string of symbols as $\{1, 3, x, 6\}$, and so $\{1, 3, x, 6\}$ is not a literal expression.

Example 1.6.6. The expression $1 + 2$ is not a literal, since $1 + 2$ evaluates to 3, which is not the same string of symbols as $1 + 2$.

Example 1.6.7. Not all constants are literal expressions. Take for example the programmer who defines the constant PI as representing the number 3.1416. Then `PI` is *not* a literal since the string of characters 'P' followed by 'I' is an invalid syntax for representing a real number. On the other hand, the string of symbols 3.1416 is valid syntax, and so 3.1416 is a literal.

1.7 Exercises

1. Which of the following pairs of sets are equal?

   a. $\{1,3,3,3,5,5,5,5,5\}$, $\{1,3,5\}$

   b. $\{\{1\}\}$, $\{1,\{1\}\}$

   c. $\emptyset$, $\{\emptyset\}$

2. Use list notation to express the set of positive integers that are perfect squares and do not exceed 100.

3. Use list notation to express the set of nonnegative integers that are less than 7.

4. Use list notation to express the set of rational numbers x for which $x^2 = 2$.

5. Use informal list notation to express the set of fractions of the form $1/n$, where n is an even positive integer between 1 and 1000 (inclusive).

6. Prove that the set of positive integers divisible by either 2 or 3 is countably infinite.

7. Prove that if A and B are countably infinite sets, then so is $A \cup B$.

8. Prove that if A and B are countably infinite sets, then so is $A \times B$. Hint: describe a dovetailing procedure similar to that used to show that the set of rational numbers is countably infinite.

9. Recall the dovetailing procedure described in Example 1.1.3. In what round will $59/457$ be listed?

10. Recall the dovetailing procedure described in Example 1.1.3. What fraction will be the 1000th listed?

11. Given $A = \{2, 4, 6\}$, $B = \{2, 6\}$, $C = \{4, 6\}$, and $D = \{4, 6, 8\}$, consider each pair of sets and determine which, if any, is a subset of the other. For example is A a subset of B? B of A? etc..

12. List all the subsets of the set $\{1, 2, 3, 4\}$.

13. List all the subsets of $\{\emptyset, \{a\}\}$.

14. How many elements are in $\mathcal{P}(A)$, where A is the set from Exercise 2?

15. Let $A = \{1, 2, 3, 4, 5\}$, and $B = \{0, 3, 6\}$. Find $A \cup B$, $A \cap B$, $A \oplus B$, $A - B$, and $B - A$.

16. Let $A = \{a, b, c, d, e\}$, and $B = \{a, b, c, d, e, f, g, h\}$. Find $A \cup B$, $A \cap B$, $A \oplus B$, $A - B$, and $B - A$.

17. Let A be the set of nonnegative integers divisible by 2, while B is the set of nonnegative integers divisible by 3. Use set-builder notation to express $A \cup B$, $A \cap B$, $A \oplus B$, $A - B$, and $B - A$.

18. Let $A_i = \{1, 2, \ldots, i\}$, for $i = 1, 2, 3, \ldots$. Use list notation to express

$$\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \cdots \cup A_n,$$

and

$$\bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap \cdots \cap A_n,$$

19. Repeat the previous problem, but now assume $A_i = \{\ldots, -2, -1, 0, 1, 2, \ldots, i\}$, for $i = 1, 2, 3, \ldots$.

20. Find $A \times B$ and $B \times A$, where $A = \{a, b, c, d\}$, and $B = \{y, z\}$.

21. Provide sets A, B, and C for which $A \times (B \times C) \neq (A \times B) \times C$. Justify your conclusion.

22. Show in general (i.e. not by a specific example) that if A and B are nonempty, and $A \neq B$, then $A \times B \neq B \times A$.

23. Let E denote the set of English words that appear in your textbook, while, P denotes the set of pages. Give an estimate of the size of $E \times P$. Give an estimate of the size of the subset A of $E \times P$, where $(w, p) \in A$ iff word w appears on page p.

24. Use a membership table to prove that the $\oplus$ operation is associative, i.e.
$$A \oplus (B \oplus C) = (A \oplus B) \oplus C.$$

25. What can you say about the sets A and B if $A \oplus B = A$? Hint: use the previous exercise and the fact that $A \oplus A = \emptyset$ for arbitrary set A.

26. Use a membership table to verify that, for any sets A, B, and C, $(B - A) \cup (C - A) = (B \cup C) - A$.

27. Verify De Morgan's first law using a membership table.

28. Prove or disprove the following statements about arbitrary sets A, B, and C.

    a. If $A \cup C = B \cup C$, then $A = B$.
    b. If $A \cap C = B \cap C$, then $A = B$.
    c. If $A \cup C = B \cup C$ and $A \cap C = B \cap C$, then $A = B$.

1.8 Exercise Solutions

1. Which of the following pairs of sets are equal?

   a. $\{1,3,3,3,5,5,5,5,5\} = \{1,3,5\}$ since each set member is only counted once; i.e. both sets are assumed to contain exactly the numbers 1, 3, and 5.

   b. $\{\{1\}\} \neq \{1,\{1\}\}$ since the first set has one member, while the second has two.

   c. $\emptyset \neq \{\emptyset\}$ since the first set has no members, while the second has one, namely the empty set.

2. $\{1,4,9,16,25,36,49,64,81,100\}$

3. $\{1,2,3,4,5,6,7\}$

4. $\emptyset$

5. $\{1/2,1/4,1/6,\ldots,1/1000\}$

6. $\{2,3,4,6,8,9,10,12,14,15,\ldots\}$.

7. Let a_1,a_2,a_3 be a listing of members of A, and b_1,b_2,b_3 be a listing of members of B. Then $a_1,b_1,a_2,b_2,a_3,b_3,\ldots$ is a listing of members of $A \cup B$.

8. Let a_1,a_2,a_3 be a listing of members of A, and b_1,b_2,b_3 be a listing of members of B.

   Round 1. Add (a_1,b_1) to the list.

   Round 2. Add (a_1,b_2), and (a_2,b_1) to the list.

   Round 3. Add (a_1,b_3), (a_2,b_2), and (a_3,b_1) to the list.

   Round $i \geq 4$. Add (a_1,b_i), (a_2,b_{i-1}), (a_3,b_{i-2}), $\ldots,(a_i,b_1)$ to the list.

9. In Round 59 59/1 is listed. Moreover, in 456 additional rounds 59/457 is listed. This makes for a total of $59 + 456 = 515$ rounds.

10. We're looking for the first n for which $1 + 2 + \cdots + (n-1) + n = n(n+1)/2 \geq 1000$. This means that $n^2 + n - 2000 \geq 0$. But the equation
$$n^2 + n - 2000 = 0$$
has positive solution
$$n = -0.5 + \sqrt{7999}/2 \approx 44.22.$$
Therefore, our desired n is $n = 45$. Moreover, after 45 rounds, 1035 fractions will have been listed, and the last one listed is 45/1. Finally, moving back 34 places takes us to the 1000th listed fraction $(45 - 34)/(1 + 34) = 11/35$.

11. We have $B \subset A$, $C \subset A$, and $C \subset D$.

12. The following subset table lists all the subsets of the set $\{1, 2, 3, 4\}$.

$1 \in S$	$2 \in S$	$3 \in S$	$4 \in S$	Subset S
0	0	0	0	$\emptyset$
0	0	0	1	$\{4\}$
0	0	1	0	$\{3\}$
0	0	1	1	$\{3, 4\}$
0	1	0	0	$\{2\}$
0	1	0	1	$\{2, 4\}$
0	1	1	0	$\{2, 3\}$
0	1	1	1	$\{2, 3, 4\}$
1	0	0	0	$\{1\}$
1	0	0	1	$\{1, 4\}$
1	0	1	0	$\{1, 3\}$
1	0	1	1	$\{1, 3, 4\}$
1	1	0	0	$\{1, 2\}$
1	1	0	1	$\{1, 2, 4\}$
1	1	1	0	$\{1, 2, 3\}$
1	1	1	1	$\{1, 2, 3, 4\}$

13. The subsets of $\{\emptyset, \{a\}\}$ are $\emptyset$, $\{\emptyset\}$, $\{\{a\}\}$, and $\{\emptyset, \{a\}\}$.

14. $|\mathcal{P}(A)| = 2^{10} = 512$

15. $A \cup B = \{0,1,2,3,4,5,6\}$, $A \cap B = \{3\}$, $A \oplus B = \{0,1,2,4,5,6\}$, $A - B = \{1,2,4,5\}$, and $B - A = \{0,6\}$

16. $A \cup B = B$, $A \cap B = A$, $A \oplus B = \{f,g,h\}$, $A - B = \emptyset$, and $B - A = \{f,g,h\}$.

17. $A \cup B = \{x | x \bmod 2 = 0 \vee x \bmod 3 = 0\}$, $A \cap B = \{x | x \bmod 2 = 0 \wedge x \bmod 3 = 0\}$, $A \oplus B = \{x | x \bmod 2 = 0 \oplus x \bmod 3 = 0\}$, $A - B = \{x | x \bmod 2 = 0 \wedge x \bmod 3 \neq 0\}$, $B - A = \{x | x \bmod 2 \neq 0 \wedge x \bmod 3 = 0\}$.

18. We have
$$\bigcup_{i=1}^{n} A_i = \{1, 2, \ldots, n\} \text{ and } \bigcap_{i=1}^{n} A_i = \{1\}.$$

19. We have
$$\bigcup_{i=1}^{n} A_i = \{\ldots, -2, -1, 0, 1, 2, \ldots, n\} \text{ and } \bigcap_{i=1}^{n} A_i = \{\ldots, -2, -1, 0, 1\}.$$

20. $A \times B = \{(a,y),(b,y),(c,y),(d,y),(a,z),(b,z),(c,z),(d,z)\}$,
$B \times A = \{(y,a),(y,b),(y,c),(y,d),(z,a),(z,b),(z,c),(z,d)\}$.

21. Let $A = \{1\}$, $B = \{a\}$, and $C = \{z\}$. Then $(1,(a,z)) \neq ((1,a),z)$, since the first tuple has elements 1 and (a,z), while the second has elements $(1,a)$, and z. Hence, they are two different tuples.

22. Assume A and B are nonempty and $A \neq B$. Let $a \in A$ and $b \in B$ be members of A and B, such that either $a \notin B$ or $b \notin A$ is true. **Case 1.** $a \notin B$. Then $(a,b) \in A \times B$, but $(a,b) \notin B \times A$. **Case 2.** $b \notin A$. Then $(b,a) \in B \times A$, but $(b,a) \notin A \times B$. $\square$

23. The estimate equals $w \times p$, where w is your estimate of the number of words per page of your textbook, and p is the number of pages of your textbook.

24. Since its last two columns are identical, the following membership table establishes $A \oplus (B \oplus C) = (A \oplus B) \oplus C$.

$x \in A$	$x \in B$	$x \in C$	$x \in A \oplus B$	$x \in B \oplus C$	$x \in A \oplus (B \oplus C)$	$x \in (A \oplus B) \oplus C$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	0	0	0
1	0	0	1	0	1	1
1	0	1	1	1	0	0
1	1	0	0	1	0	0
1	1	1	0	0	1	1

25. We have

$$A \oplus (A \oplus B) = (A \oplus A) \oplus B = \emptyset \oplus B = B,$$

where the first equality makes use of the associativity of $\oplus$. But since $A \oplus B = A$, we also have

$$A \oplus (A \oplus B) = A \oplus A = \emptyset.$$

Therefore, $B = \emptyset$.

26. The following membership table establishes $(B - A) \cup (C - A) = (B \cup C) - A$.

			$x \in$	$x \in$	$x \in$	$x \in$	$x \in$
$x \in A$	$x \in B$	$x \in C$	$B - A$	$C - A$	$B \cup C$	$(B \cup C) - A$	$((B - A) \cup ($
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1
0	1	0	1	0	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0
1	1	0	0	0	1	0	0
1	1	1	0	0	1	0	0

27. Since the final two columns are equal, the following membership table establishes $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

$x \in A$	$x \in B$	$x \in \overline{A}$	$x \in \overline{B}$	$x \in (A \cup B)$	$x \in \overline{A \cup B}$	$x \in (\overline{A} \cap \overline{B})$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

28. Prove or disprove the following statements about arbitrary sets A, B, and C.

    a. If $A \cup C = B \cup C$, then $A = B$. Sometimes false: Let $A = \emptyset$, $B = \{1\}$, $C = \{1\}$. Then $A \cup C = B \cup C$, but $A \neq B$

    b. If $A \cap C = B \cap C$, then $A = B$. Sometimes false: Let $A = \{1, 2\}$, $B = \{1, 3\}$, $C = \{1\}$. Then $A \cap C = B \cap C$, but $A \neq B$.

    c. If $A \cup C = B \cup C$ and $A \cap C = B \cap C$, then $A = B$.

Chapter 2

Introducing Clara

The goal of this chapter is to get the reader up to speed with the Clara constraint-programming language that is used throughout the remainder of the book. The name “Clara” derives from the word “clarity” and represents the idea that, before a problem can be solved, one must first have clarity about all the domains, variables, and constraints that comprise the problem.

2.1 Overview of the Clara Language and System

At the very least, a constraint-programming system consists of three parts: i) a formal language that is used for modeling problems, ii) a language interpreter that is capable to processing language statements for the purposes of analysis and evaluation, and iii) a solver that is capable of finding solutions to the problem model. We discuss all three of these in relation to

Clara.

2.1.1 The Clara language

The Clara language was designed with the following traits in mind.

Highly Expressive It allows most constraints to be expressed using common mathematical functions, thus requiring a minimal amount of translation effort in going from a natural-language statement to a formal constraint statement.

Solver Independent Modeling requires little to no knowledge of how a problem model will be processed in order to find a solution, or how a solution to a model will be found.

Function Oriented Every programming statement is a valid functional expression, meaning that i) functions are at the core of the language, ii) each statement is capable of being evaluated, and iii) program statements may be parsed in the same manner as mathematical expressions.

Extensible It allows the programmer to define new functions and data structures that are suitable for modeling problems in almost any domain of application.

Readable Although every statement is a functional expression, formatting conventions help make a program appear more readable.

Using a language that possesses the above traits seems best for learning constraint programming, since it means fewer *language constraints* and hence more freedom to model a problem in a way that seems fitting to how the problem is being perceived by the programmer.

2.1.2 The Clara interpreter

Generally speaking, programming languages can be divided into those that are compiled, and those that are interpreted. To understand the difference, recall that the CPU of a computer is designed to execute a specific set of low-level machine instructions. Moreover, the set of all possible machine instructions in itself represents a programming language, call it $\mathcal{L}$. However, this language seems quite primitive and very few programmers would enjoy programming with it. For this reason, more sophisticated higher-level languages have been designed that make programming seem much more pleasant. Let $\mathcal{H}$ denote such a high-level language. Then a **compiler** is a piece of software that translates a program written in $\mathcal{H}$ to one written in $\mathcal{L}$. Thus, only one program, namely the compiler, need ever be written in $\mathcal{L}$, while all other programs may be written in $\mathcal{H}$.

Rather than compile an $\mathcal{H}$-program P_H to an $\mathcal{L}$-program P_L, another option is to use an **interpreter** program P_M that is written in some third (usually compiled) language $\mathcal{M}$, and is capable of parsing and executing the statements of P_H in order to produce the same output/effects that would have been produced had P_H been translated to P_L. Thus, for all practical purposes, there is no need to compile P_H since P_M knows how to mimic the behavior of P_H had it been compiled to P_L.

The Clara language interpreter works by parsing and evaluating each Clara statement that it encounters. The official interpreter consists of a terminal that allows the user to type expressions on a line that begins with a prompt, such as $>$. Section 2.3 and subsequent sections explain this interpreter in detail.

2.1.3 The Clara Solver

The final part of a constraint-programming system is software that is used for finding solutions to problems that are modeled with the system's programming language. Most often, as in the case of Clara's interpreter, the solver exists as part of the interpreter. For example, the interpreter may parse an expression, such as `solve M`, and then passes control to the solver which processes model `M` for the purpose of finding a solution.

Clara's solver works by first translating each model constraint to a propositional formula. The collection of these constraint formulas, together with the variables on which they depend, forms an instance of what is called the **Satisfiability problem (SAT)**, which is the problem of finding a variable assignment that satisfies each of the formulas. Although there is no known algorithm that works efficiently on all instances of SAT, there are several known algorithms that work fairly well in practice so long as the problem instances do not grow too large in size. Clara's main solver is based on the award-winning MiniSAT solver developed by Niklas Eén and Niklas Sörensson. A vanilla version of the solver is available for free download at `minisat.se`. The MiniSAT algorithm [5] is based on fundamental problem-solving concepts, such as backtracking, constraint learning and propagation, and variable weighting. Some of these concepts will be discussed in a later chapter.

It should be noted in passing that the strength found in using a SAT solver lies in the fact that most problems of interest can be readily translated into an instance of SAT, and hence most problems of interest can in theory be solved using a SAT solver. However, there are many kinds of problems that can be solved far more efficiently using a different algorithm. For example, a problem whose constraints amount to a system of linear equations with real-valued variables can be solve far more efficiently using numerical methods, such as the Gaussian Elimination algorithm [2]. In a perfect world, before translating a problem to an instance of SAT one would first check if the problem's structure matched that of any that is solvable by some more efficient algorithm $\mathcal{A}$, in which case the problem would be solved using $\mathcal{A}$. With that said, it is important to emphasize that most problems in practice may only be *approximated* with a model that is solvable in some provably efficient way, and finding a suitable approximation model often requires significant mathematical skill and knowledge. On the other hand, modeling a problem "as is" and using a SAT solver is quite often sufficient for the needs of the specific situation. Although there is no "one size fits all" method to problem solving, SAT solving has been shown to fit comfortably on a host of interesting industrial and scientific problems [10].

2.2 Installing the Clara interpreter

The first step is to install the latest version of the Clara interpreter. For a Windows user this amounts to obtaining a copy of the `clara.exe` executable from the website, saving it in a directory. Then open a DOS command shell and use the `cd` command to navigate to the directory were the executable is saved.

2.3 The Command Terminal

Now that you have the interpreter, the next step is to begin writing and evaluating Clara expressions. The interpreter offers a command line that begins with a prompt whose default is $>$. Following the prompt, single-line expression may be entered and evaluated. For example, on the command line below, expression $1 + 1$ is entered, and its evaluation `2` is displayed below.

```
> 1+1

2
```

Another example is

```
> "Hello World!"

Hello World!
```

where the expression consists of a string literal that is echoed as output without the double quotes.

Command-line evaluation takes places within the global **interpreter environment**. This environment maintains all known program artifacts that either exist by default, or have been defined by the user. Such artifacts include sets, functions, global variables, constants, and models.

A command-line expression may make use of any standard or user-defined function. A **standard function** is a pre-defined function that has already been defined and programmed within the interpreter. On the other hand, a **user-defined function** is one that is defined by you, or some other programmer, and whose definition usually exists within a text file that is loaded into the interpreter.

One important family of standard functions is the **command** functions. These functions help you get things done within the interpreter environment. For example, the `help` command function takes zero inputs, and returns a formatted string that describes each of the command functions. Most of these functions are introduced in the following sections. Before doing so, another one we'll introduce here is the `quit` function, which also takes zero inputs, but has the effect of exiting the interpreter.

2.3.1 Keeping track of history

When using the interpreter, you'll often need to evaluate the same expression several times during your session. The simplest way to call up past expressions is to use the `up` and `down` arrow keys on your keyboard. Each pressing of the `up` key shows a previously used expression, starting with the most recently used expression, and working backwards towards the earliest one. The `down` key is similar, except the order is reversed from earliest to most recent. However, there may be times when what you're looking for falls somewhere in the middle of a long list of past expressions. In this case you can use the `hist` command that displays all past expressions. For example,

```
> hist

1) 1+1
2) "Hello World!"
3) help
```

lists all my past expressions. Then to re-evaluate a past expression, I may type the number of that expression, followed by a semicolon. For example,

```
> 3;
```

causes `getwd` to be re-evaluated.

A third way that past expressions (and their evaluations) are remembered is by use of a log file which is automatically created upon interpreter start up. To switch to a new log file, the `logfile` command takes as input a string that represents the desired filename, and creates the file `filename.log` within the working directory. Once created, every subsequent evaluated expression is stored in this file, along with the result of its evaluation. For example, after the command

```
logfile "test.log"
```

I had the interpreter evaluate the expressions 1 + 1 and “Hello World!”. Then I opened the file to find the following.

```
/*
test.log
Time of creation: Fri Jul 31 11:08:59 2020
*/


//success


1+1

//2


"Hello World!"

//Hello World!
```

The file begins by printing its name and the date and time of its creation. Notice how this information appears within the left delimiter $/*$, and the right delimiter $*/$. Text appearing within these delimiters is referred to as a **multiline comment**. A comment is any string of text that appears in the file in order to provide meta information about the file and its contents. Although the multiline delimiters $/*$ and $*/$ may be used on a single line, **single-line** comments can be delimited by two consecutive forward slashes //. In the above log file these are used to represent the expression evaluations as comments. This is because the uncommented text in a log file should be comprised only of expressions that were input on the command line in case the user wants to re-evaluate those expressions by loading the file into the interpreter. This is especially useful if you run into a problem with the interpreter and want to share the sequence of expressions that produced the problem.

If for some reason you no longer want command-line expressions recorded in the log file, then the `nolog` command prevents future expressions from being recorded. However, it does not erase any of your existing log files. Nor does switching from one log file to another delete the previous log file.

2.3.2 Loading files

Although the command-line is useful for getting things done, it is usually not the place to write a constraint program. Constraint programs can have tens, hundreds, or even thousands of lines, and so are best authored within a text editor. In general, the `load` command takes as input the name (as a string) of the file (including its extension) to be loaded. Furthermore, the file's (uncommented) lines should each consist of either an expression, or the continuation of an expression from a previous line (Section 2.7.2 discusses the formatting of multiline expressions). The interpreter then reads the file and evaluates each of its expressions one at a time from top to bottom.

Note that the default is to *silently* evaluate each expression, meaning that each expression's evaluation is not displayed on the terminal. To print the evaluations of a file expression that is being evaluated, the standard `write` function will prove helpful. Function `write` takes any item (inlcuding a string) and writes to the console, or to a file. For example, suppose that within the text file variable x has been assigned, then the statement

```
write x
```

will print the contents of x to the terminal.

Loaded files often include statements for adding program artifacts to the environment, such as constants, variables, functions, etc.. After loading such a file, there may come a time when you no longer want these artifacts in the environment. Using the `unload` command with the filename as input string has the effect of removing all artifacts that were added when this file was loaded. However, be warned that it also removes any artifacts that were added *after* these artifacts, in order to avoid any possible dependency issues. Another way to start fresh is to use the `reset` command that has the effect of removing all user-defined artifacts.

2.4 Standard Operations

Tables 2.1, 2.2, 2.3, 2.4 provide the symbols used for the different arithmetic, Boolean, set, and relational operations.

Example 2.4.1. Consider the statement

$x \in \{1,3,4,7,8,10\} \to (x{+}1 \in \{2,4,5\} \vee 2x \in \{14,16,20\}) \wedge x \notin \{-5,-3,0\}$,

Arithmetic Operation	**Clara Representation**
$+$	$+$
$-$	$-$
$\times$	$*$
$\div$	$/$
mod	%

Table 2.1: Representing arithmetic operations in Clara

Boolean Operation	**Clara Representation**
$\neg$	!
$\wedge$	&
$\vee$	$\vert$
$\oplus$	ˆ
$\rightarrow$	$->$
$\leftrightarrow$	$<->$

Table 2.2: Representing Boolean operations in Clara

where x is some variable (variables are covered in the next section). Then this statement may be entered into the interpreter as

```
> x @ {1,3,4,7,8,10} -> (x+1 @ {2,4,5} | 2*x @ {14,16,20}) &
     !(x @ {-5,-3,0} //continuation of the previous line
```

$\square$

Notice that there is no operation for exponentiating. Instead, we use the two standard functions `pow` and `root`. For example, 3.14^{-2} is written as `pow`$(3.14, -2)$, while $\sqrt{5}$ is written as `root`$(5, 2)$. In the case of `root`, the base must be nonnegative, while the integer root must must be nonzero.

Set Operation	**Clara Representation**
$\cap$	$*$
$\cup$	$+$
$-$	$-$
$\oplus$	%

Table 2.3: Representing set operations in Clara

Relational Operation	**Clara Representation**
$<$	$<$
$\leq$	$<=$
$>$	$>$
$\geq$	$>=$
$=$	$==$
$\neq$	$!=$
$\subseteq$	$<==$
$\in$	@

Table 2.4: Representing relational operations in Clara

Two useful Boolean functions are `and` and `or`, the prefix versions of the AND and OR operations. Both accept one or more Boolean expressions. For example,

```
> and(x,y,z,true,true,false,w)

false
```

evaluates to `false` because the 6th input is `false` which, along with `true`, is a standard Boolean constant.

Finally, the `size` function may be used to determine the cardinality of a set. For example, the expressions

```
> size {1,3,4,7,8,10}

6

> size { }

0
```

evaluate to 6 and 0 as expected.

Note that, for operations requiring more than one character, such as <=, no whitespace is allowed between consecutive characters. Also, there should be whitespace between the character(s) of an operation and the negative sign in front of a number. One reason for this is that the user is allowed to define new operations, such as $+-$, and so we wouldn't want to confuse $3 +- 5$ with $3 + (-5)$.

2.4.1 The `scope` functions

Recall from Section 1.6.3 that, within an expression, both arithmetic and set operations take precedence over relational operations, which in turn take precedence over Boolean operations. This means, e.g., that, in the absence of parentheses, arithmetic and set operations get performed first, followed by relational and Boolean operations. In reality, in the absence of parentheses the interpreter uses the standard `scope` function to determine which operation takes precedence. This function takes as input a function f, and outputs a positive integer. Moreover, the lower the value of `scope(f)`, the higher the precedence of f. Thus, in the absence of parentheses, the operation having highest scope will be the last to be evaluated. The idea behind this is that the final operation op serves as the root of the expression's parse tree, and so op has the greatest *visual scope*, since there is a downward-directed path from op to every other node in the tree. A few notes about the `scope` function are now given.

1. All postfix functions have the lowest scope at 1000.
2. All prefix functions have the second lowest scope at 1100. Furthermore, if f is a prefix function with n inputs, then the expression

   $$f\ e_1, e_2, \ldots, e_n$$

   is parsed in the same manner as

   $$f(e_1, e_2, \ldots, e_n).$$

   In other words, the parser always assumes that included with a prefix function is its pair of input-delimiting parentheses. This makes sense because f should be at the root of the parse tree, and evaluated only after each expression e_i, $i = 1, \ldots, n$, has been evaluated.
3. Infix functions that have the same name also have the same scope. For example, 2300 is the scope for both the arithmetic mod function, and the symmetric-difference set function, since both are named %.
4. In the case that an infix function is defined (by the programmer) using a unique name, then its default scope is determined by its first character. For this reason the programmer should use another version of `scope` that takes two inputs: i) the newly defined infix function f having the unique name, and ii) a positive integer s representing the desired scope. Finally, this `void` version of `scope` has the effect of assigning scope s to f.

2.5 Constants and Variables

In this section we examine how to define constants and variables within the interpreter environment.

From Chapter 1, recall that a variable v consists of a name, and a domain set of possible values that may be assigned to v. For example, the variable named `height` may have a domain set of all positive integers from 1 to 100.

In Clara, a **constant** is defined as a special kind of variable, i.e. it has a name, and its domain set consists of exactly one value to which it is always assigned. This definition differs from the mathematical one since, in mathematics, 3.14 is considered a constant, but not so in Clara because there isn't a name associated with this value. In programming we think of 3.14 as a literal (see Chapter 1), i.e. an expression whose evaluation is equal to itself: eval(3.14) = 3.14. On the other hand, the constant named `PI` that is assigned the value 3.14 is not a literal since eval(`PI`) = 3.14, which is not syntactically equal to `PI`.

In programming, the term **identifier** means a sequence of characters that is used to represent the name of some programming artifact, such as a constant, variable, function, etc.. In this section we focus on identifiers that are used to name constants and variables. Such identifiers must follow one of the following naming conventions.

Lowercase Naming Convention The first character is a lowercase letter, the last character is alphanumeric, and all characters in between must be alphanumeric or underscore.

Uppercase Naming Convention The first character is an uppercase letter, the last character is alphanumeric, and all characters in between must be numeric, an uppercase letter, or underscore.

Examples of legal variable names include `x`, `t_11`, `camelCaps`, `file_name`, `PI`, and `MAX_ITERATIONS`, while names such as `7eleven`, `ComplexNum`, `_X11`, and `PI_` are illegal as a variable name.

2.5.1 The `const` function

The standard `const` function takes two inputs, an identifier and a value, and has the effect of creating a constant and registering it in the interpreter environment. For example,

```
> const(PI,3.14)
```

creates the constant `PI` whose assigned value is 3.14, and registers it in the evironment. Now all subsequent expressions may use this constant, as in

```
> 2*PI*1

6.28
```

As mentioned in Section 2.1, the Clara language has formatting conventions for enhancing the readability of expressions. One such formatting convention is to allow a prefix function that begins a line to drop its input parentheses. For example, the above expression that defined `PI` could have also been written as

```
> const PI, 3.14
```

The most common mistake when using this convention is to forget the commas that separate each input, which will produce an error (try it).

2.5.2 Set constants

There are several standard constants that represent sets of things. The following is a description of each one.

Any the interpreter's universal set

Boolean the set {`false`, `true`}

Character the set of ASCII characters

Integer the set of double precision floating-point numbers x that satisfy the predicate $x = \lfloor x \rfloor$.

Real the set of double precision floating-point numbers

String the set of all strings

Identifier the set of all identifiers

Tuple the set of all tuples

Map the set of all maps

Matrix the set of all matrices

Model the set of all members of all defined model sets

Objective the set of all objectives

Set the set of all sets

Multiset the set of all multisets

Function the set of all functions

Expression the set of all expressions

Void the singleton set {`void`} consisting of `void`

2.5.3 The `var` function

The standard `var` function takes two or more inputs, where the final input is a set S, and the other inputs are variable identifiers. The effect of this function is to create a number of variables, one for each identifier, and each having S as its domain. For example,

```
> var a,b,c, Boolean
```

creates three Boolean variables a, b, and c. All variables created with `var` are *global*, meaning that they may be used in any subsequent expression. To assign a variable we use the assignment operation :=. For example,

```
> a := true

success
```

assigns the value `true` to Boolean variable a. A common mistake when assigning a variable is using = instead of :=. For example,

```
> a = true

a = true
```

echoes itself as output. This is because = is the **option** operation, and is used for specifying an optional input value for a function. For example, the standard `solve` function allows the option of specifying how much time the solver may take for finding a solution. Thus,

```
> solve M, time = 60
```

allows the solver to take up to 60 seconds to find a solution for model M (more on `solve` shortly). Finally, the option operation outputs an actual option, which explains the echoing of `a = true`, since this is how an option is written as a string. We'll show other uses for = in later sections. For now, notice that the mistake of using = instead of := usually does not produce an error, which can lead to a false sense of correctness and subtle errors that may take a while to diagnose. So be careful!

One useful tip here is that a variable does not have to be formally defined before using it. For example, instead of first defining variable a as a Boolean, my first statement could have been

```
> a := true
```

which has the effect of creating variable a and assigning it the value `true`. However, a's domain does *not* equal `Boolean`, but rather is equal to `Any`, which is the set of all possible things. Because of this, a can be assigned anything under the sun, including a numerical, string, or tuple value. So why would anyone want to limit the power of a variable by restricting its domain via a `var` definition? There are a few important reasons that will be explained in later chapters.

2.5.4 Standard variables

A **standard** variable is one that is already pre-defined within the interpreter. Each of these variables controls some aspect of how the interpreter behaves. As an example, the value assigned to standard variable `DECIMAL_PLACES` indicates the number of digits to the right of the decimal that are displayed when writing a floating point number to the screen or to a file. The value of the floating point number is rounded to this number of places. For example,

```
> sin 1.5

1.00

> DECIMAL_PLACES := 7

success

> sin 1.5

0.9974950
```

shows how $\sin 1.5$ is displayed using first two and then seven decimal places. It's worth noting that all trig functions only accept angles measured in radians. A listing and explanation of each standard variable may be found in the appendix.

2.6 Synatax Errors and Exceptions

When beginning the study and practice of a new programming language, the number of errors you encounter may seem overwhelming at times. Part of the job of an interpreter is to identify some of these errors and explain them in a hopefully comprehensible way.

2.6.1 Syntax errors

The most common kind of error is the **synatax error**, and is the result of not following the rules set in place for writing expressions. For example, one rule states that, if e is an expression, then so is (e). On the other hand, $(e$ is not a legal expression because it's missing a right parenthesis, and so the intepreter identifies this error as follows.

```
> (4 + 5

Error line 1: there is a left parenthesis,
but no matching right parenthesis.
```

Fortunately, there are only a few rules that govern the formation of an expression, and you may find it helpful to review them in Section 1.6. It's worth noting that not all errors that arise from the creation of an expression are syntax-related. For example, the expression

$$(p \wedge q) \div (q \vee r)$$

follows correct syntax, but is invalid since both $p \wedge q$ and $q \vee r$ evaluate to Boolean values, but $\div$ operates on numbers and Boolean values are not necessarily numerical. And although in this book we sometimes for the sake of simplicity use, say, 0 in place of `false`, Boolean values have an official non-numerical status. However, we may compute the above expression using the interpreter with the help of the standard `toreal` function that inputs the Boolean value `false` (respectively `true`) and outputs 0 (respectively, 1). For example, if a and b are assigned `true`, while c is assigned `false`, then

```
> toreal(a & b) / toreal(a | b)
```

evaluates to $1/1 = 1$.

2.6.2 Exceptions

For practical reasons, many of the standard functions have domain sets that include members that should not be input into the function. For example, the division operation / has two domain sets, both equal to `Real`, the set of real numbers. However, not only is there a limit to the size of real numbers that can serve as input, but dividing by zero is undefined. The latter produces an **exception error**, i.e. a kind of error that occurs when a function is asked to work on an input that it was not designed to handle. For example, we get the following message when dividing by zero within the interpreter.

```
> 1 / 0

An exception occurred while evaluating expression 1 / 0.
Exception thrown by function /(Real,3) on inputs 1,0.
Exception message: a divisor must be nonzero.
```

The message indicates the expression being evaluated, the function within the expression that produced the exception, and the reason for the exception.

Like the handyman's slogan "no job is too big or too small" a function's advertised domain will often have exceptions. So why not write the domain in the form $D - E$, where D is the domain set (including exceptions), and E is the set of exceptions. Although this can be done for some functions, the set E may seem difficult to express for others. Thus, when describing a function, in addition to providing its domain and range, and a general description of what it does, it's also important to describe all exceptions that may arise. In a later chapter we introduce the `assert` function that allows the interpreter to automatically check if a function's inputs meet a more refined set of criteria so as to guarantee not causing an exception.

2.7 Defining Models

Consider the following logic problem. Zowie was wondering which of her five friends, Alice, Bob, Cathy, Debra, and Evan would attend her birthday party. She knows that if Alice attends, then either Bob or Evan will attend (or possibly both), since one of them is dating Alice. Moreover, either Bob or Debra will attend, but not both, since they both work at the same coffee shop, and one of them must cover the afternoon shift. If Cathy attends, then both Alice and Bob will attend, since Alice and Bob are her closest friends. Furthermore, either Alice will not attend or Bob will not attend if Debra does not attend. Finally, Zowie knows that either Alice or Cathy (or both) will attend. Determine which of the five friends are sure to attend, and which are sure not to attend.

Before continuing, you may want to try solving this problem on your own.

One way of solving it is to represent each clue as a propositional formula that depends on one or more of the Boolean variables a, b, c, d, e where, e.g., the truth value assigned to a indicates whether or not Alice will attend the party. Then construct truth tables for each formula, and visually look for a variable assignment that yields a `true` value in every table. In what follows we show how to model this problem as a constraint program, and solve it through the interpreter.

The first step towards modeling this problem as a constraint program is to identify the variables on which it depends. In this case we use the aforementioned variables a, b, c, d, e. The next step is to collect these variables within a data structure. This is accomplished by by the following statement.

```
> model PartyPuzzle, Boolean:a, Boolean:b, Boolean:c,
     Boolean:d, Boolean:e
```

The standard `model` function takes two or more inputs and returns a model set. The first input to `model` is an identifier that gives the name of the model being created. This name must follow the **set-naming convention** (yes, models are sets!), meaning that it must begin with a capital letter, and consist only of alphanumeric characters (no underscores). In this case the model being created is called `PartyPuzzle`. The remaining input(s) to `model` consists of one or more variables, where each variable is referred to as a **model attribute**. In this case we're adding five Boolean attributes to the `PartyPuzzle`. This is done using the variable (i.e. colon) operation : whose left input is a set, the domain of the variable being defined, and whose right input is an identifier that gives its name. For example `Boolean : a` outputs a Boolean variable named a. Now, we've already seen how the `var` function is also used to create variables. The difference is that a variable created using `var` gets registered as a global variable, where as a variable created by : is transient, unless it is being used in some way so as to create an artifact that gets registered in the environment. In fact, that's exactly what is happening here: Boolean variable a is being used to create an attribute for a model that is to be registered.

Before continuing with the creation of our model, it's worth noting that there is another version of : whose only difference is that the right input is a tuple of identifiers, and whose output is a tuple of variables (actually, even the original version of : outputs a tuple having just one variable). Indeed, our model could have been created using

```
> model PartyPuzzle, Boolean:(a,b,c,d,e)
```

which appears more organized, and involves far less typing.

Our next step is to place constraints on the attribute variables. This can be done using the standard `constraint` function, whose first input is a model identifier, and whose remaining inputs each consist of a Boolean-valued expression that depends on the model attributes. For example, Zowie knows that Alice or Cathy (or both) will attend her party. Therefore, we may write

```
> constraint PartyPuzzle, a | c
```

which has the effect of adding the constraint $a \vee c$ to `PartyPuzzle`. Moreover, if we analyze the statments made in the puzzle, we see that there are four other constraints that can be added: $a \to (b \vee e)$, $b \oplus d$, $c \to (a \wedge b)$, and $\neg d \to (\neg a \vee \neg b)$. Thus we should write the expression

```
> constraint PartyPuzzle, a -> (b | d), b ^ d, c -> (a & b),
    !d -> (!a | !b)
```

2.7.1 Solving the puzzle

We are now in a position to find a solution to `PartyPuzzle`, i.e. to find an assignment for the attributes $a, b, \ldots, e$ so that each constraint evaluates to `true`. This can be accomplished using the standard `solve` function whose only required input is a model identifier. For example,

```
> solve PartyPuzzle

new_PartyPuzzle(a=true,b=false,c=false,d=true,e=true)
```

indicates that the solver found a member of `PartyPuzzle` that satisfies all constraints, namely the member whose attribute values are $a =$ `true`, $b =$ `false`, $c =$ `false`, $d =$ `true`, $e =$ `true`. Is this the only solution? To find out we may use the standard `multisolve` function, which is similar to `solve` except i) it requires an additional positive-integer input that indicates how many solutions are sought, and ii) it outputs a set of `PartyPuzzle` members, rather than a single member. For example,

```
> multisolve PartyPuzzle, 2

{new_PartyPuzzle(a=true,b=false,c=false,d=true,e=true)}
```

shows that only one solution was found, namely the solution found using `solve`. Does this mean there are no more solutions? May be. It's possible that another solution exists, but the solver ran out of time when searching for it. Both `solve` and `multisolve` have default search times equal to five seconds. To remove all doubt, we could use the optional `complete` input to ensure that the solver does not stop its search until it has exhausted all possibilities. For example,

```
> multisolve PartyPuzzle, 2, complete = true

{new_PartyPuzzle(a=true,b=false,c=false,d=true,e=true)}
```

indicates that there is indeed a unique solution.

2.7.2 Defining models within a text file

Using the interpreter's command line may seem acceptable for defining small models, but can seem cumbersome for larger ones. Recall that the `load` function allows the interpreter to evaluate one-by-one batches of expressions appearing in the same text file. Using a text file offers the following advantages.

Persistence Statements can be saved and reused at a later time.

Organization Statements that work together in order to define a programming artifact, such as a model or function, can be grouped together in the file.

Readability Text files allow for the use of comments, and allow for long expressions to be written on multiple lines.

The use of comments within a file was discussed in Section 2.3. We now show how to write an expression using multiple lines. This technique for improving readability requires that each line begin with zero or more tab spaces, with the number of tabs being referred to as the **tab number** for that line. More precisely, the tab number for a line is the number of tab characters that precede the first non-whitespace character of the line. For example, a line that begins with the character sequence

```
  \t\t     \t if i >= 0, \t x := 1
```

has a tab number equal to three.

Given the definition of tab number, we are now ready to describe a rule for using multiple lines to write an expression. Recall that the input parentheses for a prefix function f that begins a line may be dropped. In addition to this, the inputs to f may be written on successive lines, called **continuation lines**, where the tab number of these lines is one greater than the tab number of the **root line**, i.e. the line on which the name of f is written. It should be emphasized that only *one* input is allowed per continuation line, and that not all the inputs have to appear on a continuation line. For example, the line

```
and a | b, a & !d, c -> a | !d, a ^ c
```

may be written as

```
and
    a | b
    a & !d
    c -> a | !d
    a ^ c
```

or as

```
and a | b, a & !d
    c -> a | !d
    a ^ c
```

Notice how a new line replaces the need to use a comma in separating one input from the next one.

Example 2.7.1. Expression continuation may be done recursively, in that a continuation line may itself be the root line of an expression continuation. To see this, the single-line expression (written with two lines to stay with in the margins of this book)

```
and c <-> a, or(a | b, a & !d),
     or(c -> a | !d, a ^ c), e -> !a & !c
```

has a valid multiline representation of

```
and
     c <-> a
     or a | b
          a & !d
     or c -> a | !d
          a ^ c
     e -> !a & !c
```

Notice that the root line beginning with `and` has a tab number equal to zero, while each of its four input lines has a tab number equal to one. Moreover, two of those inputs make use of expression continuation, and their continuation lines have tab numbers equal to two, and hence they are each inputs to their respective `or` functions. If their tab numers had been equal to one, then they would have been treated as inputs to `and`, rather than `or`. □

Now let's apply line continuation to the `PartyPuzzle` model. After appropriate tabbing of input lines we get the following.

```
model PartyPuzzle
     Boolean:(a,b,c,d,e)

constraint PartyPuzzle
      //C1: if Alice attends, then either Bob or Evan attend
      a -> (b | e)

      //C2: either Bob or Debra will attend, but not both
      b^d

      //C3: if Cathy attends, then Alice and Bob attend
      c -> (a & b)

      /*
      C4: either Alice will not attend or Bob will not attend
           if Debra does not attend
      */
      !d -> (!a | !b)

      //C5: Alice or Cathy (or both) will attend
      a | c
```

Finally, an expression may continue on the next line if the line ends with either a comma, or infix operation, and so long as the continuation lines have a tab number equal to one greater than the originating root line. For example,

```
a | b ->
     c & !d &
     e & f
```

is one way of writing the Boolean expression

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

2.7.3 Formatting model solutions

When the interpreter displays the evaluation of an expression, the default way to display is to use the standard `expression_string` function. This function accpets any item as input, and outputs a string represents the item in a syntactically-correct way in accordance with the Clara language. For example,

```
> expression_string("Hello World!\n")

"Hello World!\n"
```

since the correct program syntax is to delimit a string using double quotes. One way to think of it is that the output of `expression_string` could in turn be parsed by the interpreter without error. On the other hand, notice that

```
> "Hello World!"

Hello World!
```

yields the same output, but without the double quotes. Thus, the interpreter would not parse this output as a string, since the necessary double quotes are missing. Furthermore, strings are one of the two exceptions to the rule of using `expression_string` to format the output. The other exception is made for members of model sets. Indeed, both string and model-set members are formatted with the standard `tostring` function. When `tostring` is applied to a string input *s*, it displays *s* without the double quotes, and correctly displays its whitespace and escape characters, such as with the following example.

```
> "1)\tred\n2)\torange\n3)\tpurple\n\n\n\"colors!\""

1)      red
2)      orange
3)      purple


"colors!"
```

In the case of a model-set member, the default way of writing it is

```
new_ModelName(attr_1=value1,attr_2=value2,...,
      attr_n = value_n)
```

where the attributes are ordered in a manner that is consistent with the order in which they appeared in the model's definition. Unfortunately, this format can become highly impractical, such as in the case when a model solution is, say, a schedule that one wishes to display in spreadsheet software. In such cases we may define our own `tostring` function that formats the output in a more usable and readable manner. As an example, let's write a `tostring` function for `PartyPuzzle`. We want the output to indicate which friends are attending the party, and which ones are not. The first step is to define the function's signature as

```
function tostring String, PartyPuzzle:p
```

which makes use of the standard `function` function. Function `function` takes as input an identifier serving as the function's name, the codomain set, and zero or more additional variable inputs that serve as the function's domain variables. In the above use of `function`, we see that the function being named is `tostring`, its codomain is the set `String`, which is a standard constant that denotes the set of all of strings, and it has one domain variable p that accepts a `PartyPuzzle` member. Thus, we have just defined a function whose signature is

```
tostring(PartyPuzzle,String)
```

The next step is to provide a rule for converting input p to a string output. This is accomplished using the `rule` function whose first input is the name of the function to which the rule applies, and the second input is an expression that depends on the input variables, and evaluates to a member of the codomain. To write the rule's expression we'll make use of the string concatenation operation $+$ whose left and right string inputs s_1 and s_2 are combined to produce the output $s_1 s_2$, as is shown with

```
> "house" + "hold"

household
```

The other standard function we'll need is the ternary function `ifelse` whose first input is a Boolean value which, if equal to `true`, causes `ifelse` to output the second input. Otherwise it outputs the third input, as is shown with

```
> ifelse 1+1 == 2, "correct", "incorrect"

correct
```

We now have all we need for defining the rule as follows.

```
rule tostring
        "Friends attending the party: " +
            ifelse(p.a,"Alice ","") +
            ifelse(p.b,"Bob ","") +
            ifelse(p.c,"Cathy ","") +
            ifelse(p.d,"Debra ","") +
            ifelse(p.e,"Evan ","") + "\n" +
            "Friends not attending the party: " +
            ifelse(!p.a,"Alice ","") +
            ifelse(!p.b,"Bob ","") +
            ifelse(!p.c,"Cathy ","") +
            ifelse(!p.d,"Debra ","") +
            ifelse(!p.e,"Evan ","")
```

Using our new `tostring` function, we get the following more readable output.

```
> solve PartyPuzzle

Friends attending the party: Alice, Debra, Evan
Friends not attending the party: Bob, Cathy
```

A Cryptarithmetic puzzle

A cryptarithmetic puzzle is a mathematical equation whose terms are equal to blocks of letters, with each letter representing a unique digit between 0 and 9. When each letter is replaced by its digit, each block turns into a number and the numbers must balance the equation. Finally, each digit may be represented by at most one letter. For example, consider the cryptarithmetic puzzle

$$\text{SEND} + \text{MORE} = \text{MONEY}.$$

The solution to this puzzle is O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9, since

$$9567 + 1085 = 10652.$$

Let's provide a model for this problem and solve it through the interpreter. We'll start by using the eight variables s, e, n, d, m, o, r, y, and giving each a discrete domain of $D = [0, 1, \ldots, 9]$. Notice that they are lowercase, since all model attributes must be lowercase. So far we have

```
set D, 0..10

model CryptArithmetic
     D:(s,e,n,d,m,o,r,y)
```

Now for the constraints. The first constraint converts the word equation into a numerical one as follows.

```
constraint CryptArithmetic
        (1000*s + 100*e + 10*n + d) +
            (1000*m + 100*o + 10*r + e) ==
                (10000*m + 1000*o + 100*n + 10*e + y)

        m != 0 & s != 0 //no leading digit equals 0
```

Finally, we need to state that now two variables can assume the same value. To do this we use the standard `alldifferent` function that accepts any number of inputs, and return `true` iff no two inputs are identical. Hence, we may write the final constraint as

```
constraint CryptArithmetic
        alldifferent(s,e,n,d,m,o,r,y)
```

Finally,

```
> solve CryptArithmetic

new_CryptArithmetic(s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2)
```

gets us the solution. To verify $9567 + 1085$ does in fact equal 10,652.

2.8 Mixed-variable modeling

One of the advantages of constraint programming is that it allows for variables of all types to occur within the same constraint. In this next example we model and solve a puzzle that has both numerical and Boolean variables. The puzzle is stated as follows in the first person. "My constraint-programming class consists of 16 students (including myself), some of whom are math majors, and the rest of whom are computer science (cs) majors; some of whom are undergraduates, and rest of whom are graduates. The following statements are true, whether or not you include me in the head count. There are

1. more cs students than math students,
2. more graduate math students than graduate cs students,
3. more graduate cs students than undergraduate cs students,
4. and at least one undergraduate math student.

Who am I: a math or cs student? a graduate or undergraduate?"

As before, the first step is to define the variables that comprise the problem. To solve the puzzle we need to determine the number of undergraduate math (`um`), undergraduate cs (`ucs`), graduate math (`gm`), and graduate cs students (`gcs`). In all cases we can safely assign each variable a domain of $\{0, 1, \ldots, 16\}$. Furthermore, to save typing we can give this set a name using the standard `set` function whose first input is the name we want to assign the set, and the second input is the set to be named.

First let's address how to create the set $\{0, 1, \ldots, 16\}$. The long way is to simply write it as the literal

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}.$$

However, we may also use the standard interval function .. whose left and right inputs are two real numbers a and b, $a \leq b$, and whose output is the interval $[a, b]$. Now if a and b are integers, then $a..b$ evaluates to the integer interval $\{a, a+1, a+2, \ldots, b\}$. On the other hand, if either a or

b has a nonzero fractional part, then $[a, b]$ is treated as the continuous set of real numbers between a and b (note: to create a continuous interval with integers a and b, use $a :: b$). Recalling the set-naming convention that starts a set name with a capital and only uses alphanumeric characters, we'll name the set `N` with the statement

```
set N, 0..16
```

The other two variables we need are two Booleans, one (`is_math`) that indicates whether or not the speaker is a math major, and the other (`is_grad`) to indicate whether or not the speaker is a graduate student. Now we may define the model with

```
model CPClass
     N:(um,gm,ucs,gcs)
     Boolean:(is_math,is_grad)
```

and the model constraints as

```
constraint CPClass
      //C1: more cs than math
      ucs + gcs > um + gm

      //C2: more cs than math,
      //even if speaker is excluded
      !is_math -> ucs + gcs - 1 > um + gm

      //C3: more grad math than grad cs
      gm > gcs
```

```
//C4: more grad math than grad cs,
//even if speaker is excluded
is_math & is_grad -> gm - 1 > gcs

//C5: more grad cs than undergrad cs
gcs > ucs

//C6: more grad cs than undergrad cs,
//even if speaker is excluded
!is_math & is_grad -> gcs - 1 > ucs

//C7: at least one undergrad math
um > 0

//C8: at least one undergrad math,
//even if speaker is excluded
is_math & !is_grad -> um > 1

//C9: there are 16 students in the class
gcs + cs + um + gm == 16
```

The odd-numbered constraints use only the numerical variables, while the even-numbered ones use a mix of Boolean and numerical. Let's examine one of these mixed constraints, such as C2. It's an if-then statement whose hypothesis is that the speaker is a cs major. Now, if this is false, then the statement becomes vacuously true. On the other hand, suppose the speaker is a cs major. Then we must subtract one from the left side of the conclusion inequality since we want to exclude the speaker from the head count of cs majors (the speaker claims that the statement is true even if the speaker is excluded in the head count).

Solving this model we get

```
> solve CPClass

new_CPClass(um=1,gm=6,ucs=4,gcs=5,is_math=false,is_grad=false)
```

Telling us that the speaker is an undergraduate cs major.

2.9 Exercises

1. Provide a programming expression for the mathematical expression $((7x^2 - 3x) \div (2y + 7)) \bmod 3$. Assume variables x and y have already been defined.

2. Provide a programming expression for the logical expression

   $$(\neg q \wedge (p \rightarrow q)) \vee \neg p.$$

   Assume variables p and q have already been defined.

3. Provide a programming expression for the mathematical expression

   $$x \in A \oplus B \leftrightarrow x \in A - B \vee x \in B - A.$$

   Assume variables x, A, and B have already been defined.

4. Evaluate each of the following set programming expressions.

```
a. {1,4,6,10} + {2,4,5,6,9}
b. {1,4,6,10} * {2,4,5,6,9}
c. {1,4,6,10} - {2,4,5,6,9}
d. {1,4,6,10} % {2,4,5,6,9}
```

5. Provide a programming expression for the mathematical expression $x^{3/5}$. Assume variable x has already been defined.

6. In addition to Boolean, string, and numerical literals, Clara also allows for ASCII character literals that are delimited using a pair of single quotes, such as

   ```
   'H', 'i', '!', and '\n'.
   ```

   Provide a programming statement that defines the set `LVowel` of all lowercase vowels.

7. Another way to define a set of characters is to use the standard `toset` function, one version of which takes as input a string s, and returns a set whose members are the characters of s. For example,

   ```
   set LVowel, toset "aeiou"
   ```

   defines the `LVowel` set from the previous exercise. Use to set to define the set `Digit` of numerical characters.

8. Provide programming statements for defining each of the following sets.

   a. The set `Temperature` consisting of all real numbers between 0 and 1000.
   b. The set `Upper` consisting of all uppercase letters.
   c. The set `True` consisting of the single Boolean member `true`.
   d. The set `Hour` consisting of the integers $0, \ldots, 23$.
   e. The set `J` consisting of $\{0, \ldots, 50\} \cup \{100, \ldots, 150\}$
   f. The set `Person` consisting of the names "Alice", "Bram", and "Cindy".
   g. The power set P of the set $\{1, 2\}$.

9. Rewrite the following programming expression using three lines and as few parentheses as possible.

```
g((5*(3+2)),(-(5)),((3*7)+2))
```

Do this by placing the second and third inputs to g on separate (tabbed) lines.

10. Define a program model whose unique solution is the Boolean variable assignment over variables $a, b, \ldots, g$ that satisfies each of the following formulas.

    a. $d \oplus g$
    b. $g \rightarrow (\neg e \wedge d)$
    c. $a \vee f \vee \neg g$
    d. $a \leftrightarrow g$
    e. $\neg c \rightarrow (a \vee b)$
    f. $c \rightarrow (a \wedge \neg d)$
    g. $a \vee c \vee e$
    h. $b \oplus f$

11. Define a program model whose unique solution is the set variable assignment over variables r, s, t that satisfies each of the following set equations. Assume the domain of each variable is the power set of $\{1, 2, \ldots, 20\}$. Hint: the program statement for the power set of $\{1, 2, \ldots, 20\}$ is

```
Set(1..20)
```

    a. $r \cup s = \{2, 5, 8, 9, 10, 11, 13, 15, \ldots, 20\}$
    b. $r \oplus t = \{4, 5, \ldots, 13, 16, 17\}$
    c. $s \cap t = \{10\}$
    d. $r - s = \{2, 5, 8, 9, 13, 15, 16, 19\}$

12. Consider the following 4×4 **magic square** problem instance.

13			12
2			7
		4	
		15	1

The blank squares should be filled in with values from the set

$$\{1, 2, \ldots, 16\}$$

in such a way that i) no two squares hold the same value, and ii) the sum of the values held in any row, column, or main diagonal equals a constant (what must this constant equal?). Define a program model whose unique solution is the unique solution to the above magic square problem instance.

2.10 Exercise Solutions

1. We have the following.

```
((7*pow(x,2) - 3*x)/(2*y + 7)) % 3)
```

2. We have the following.

```
(!q & (p -> q)) | !p
```

3. We have the following.

```
x @ (A % B) <-> x @ (A-B) \vee x @ (B-A)
```

4. We have the following evaluations.

```
a. 1..2 + 4..6 + 9..10 (or {1,2,4,5,6,9,10})
b. {4,6}
c. {1,10}
d. 1..2 + {5} + 9..10
```

5. `pow`($`root`(x, 5), 3$)

6. We have the following.

   ```
   set LVowel, {'a','e','i','o','u'}
   ```

7. We have the following.

   ```
   set NumChar, toset "0123456789"
   ```

8. We have the following set definitions.

   ```
   a. set Temperature, 0::1000
   b. set Upper, toset "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   c. set True, {true}
   d. set Hour, 0..23
   e. set J, 0..50 + 100..150
   f. set Person, {"Alice", "Bram", and "Cindy"}
   g. set P, {{},{1},{2},{1,2}}
   ```

9. We have the following.

   ```
   g 5*(3+2)
       -5
        3*7 + 2
   ```

10. We have the following model.

    ```
    model Exercise10
            Boolean:(a,b,c,d,e,f,g)

    constraint Exercise10
        d ^ g
        g -> (!e & d)
        a | f | !g
        a <-> g
    ```

```
    !c -> a | b
    c -> a & b
    a | c | e
    b ^ f
```

In addition we have the following.

```
> solve Exercise10

new_Exercise10(a=false,b=true,c=false,
    d=true,e=true,f=false,g=false)
```

11. We have the following model.

```
model Exercise11
    Set(1..20):(r,s,t)

constraint Exercise11
    r+s == {2,5,13} + 8..11 + 15..20
    r%t == 4..13 + {16,17}
    s*t == {10}
    r-s == {2,5,8,9,13,15,16,19}
```

In addition we have the following.

```
> solve Exercise11

new_Exercise11(
    r={2,5} + 8..9 + {11,13} + 15..17 + {19},
    s=10..11 + 17..18 + {20},
    t={2,4} + 6..7 + {10,12,15,19})
```

12. Since the no two squares hold the same value, the values held in all squares sum to

$$1 + 2 + \ldots + 16 = 16(17)/2 = 136.$$

Finally, 136/4 = 34 which tells us each row sum to 34 (and so the same must be true for each column and main diagonal). We have the following model.

```
set N, 1..16

model Exercise12
    //e.g. a = value held in row 1, column 1
    //b = value held in row 1, column 2, etc.
    N:(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)

constraint Exercise12
    //rows
    a + b + c + d == 34
    e + f + g + h == 34
    i + j + k + l == 34
    m + n + o + p == 34

    //columns
    a + e + i + m == 34
    b + f + j + n == 34
    c + g + k + o == 34
    d + h + l + p == 34

    //diagonals
    a + f + k + p == 34
    m + j + g + d == 34

    //hints
    a == 13 & d == 12
    e ==  2 & h ==  7
    k == 4
    o == 15 & p == 1
    //a-p must all be different
    alldifferent(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
```

In addition we have the following.

```
> solve Exercise12

new_Exercise12(a=13,b=3,c=6,d=12,e=2,f=16,g=9,h=7,
    i=11,j=5,k=4,l=14,m=8,n=10,o=15,p=1)
```

Chapter 3

Structures and Iterators

In the previous chapter we learned how to construct a model for a given problem. It always began with defining the model attributes that serve as variables that need to be assigned in a way that satisfies each of the model constraints. Furthermore, each attribute corresponded with a single Boolean or numerical variable. This works fine for problems with only a handful of variables, but when the number of variables climbs into the hundreds, thousands, and beyond, a one-to-one correspondence between attributes and variables can lead to an unwieldy and unreadable model. Thus, we would prefer that each model attribute represent as many variables as possible so the model can be expressed with a minimum number of attributes and constraint statements.

In this chapter we introduce structures that help reduce the number of attributes needed to model a problem. These structures include tuples, maps, models, and matrices, and each one is capable of serving a dual purpose within a constraint program. Indeed, on one hand each is capable of holding data that is needed to describe a problem. When using one in this manner we call it as a **data structure**. On the other hand, each is also capable of representing an arbitrary number of problem variables, and when using one in the manner we call it a **variable structure**.

Throughout this chapter we'll see how structures work hand-in-hand with iterator functions. These functions seem quite useful for constraint programming because a single use of an iterator is capable of representing an arbitrary number of constraints, all rolled up into one statement. The most commonly used iterators are the quantifiers $\forall$ and $\exists$ from predicate logic, as well as Σ from arithmetic.

Example 3.0.1. As a preview, suppose f is a function having domain

$$\{1, \ldots, 500\},$$

where $f(i) = v_i$ assigns i to a variable v_i, whose domain is

$$\{1, \ldots, 100\},$$

for all $i = 1, 2, \ldots, 500\}$. Moreover, suppose we want v_i to satisfy the constraint $v_i + v_{i-1} = i$, for all $i \geq 2$. This can be done with a single constraint, namely

$$\forall i(i \geq 2 \to f(i) + f(i-1) = 1.$$

In this example, we see how f serves as a (tuple or array) variable structure, because it represents 500 different variables. And with the help of the $\forall$ iterator, we were able to use a single constraint to express 499 different constraints.

Later in the Chapter we'll show how to write the above family of constraints using just a few programming statements.

3.1 The Tuple Structure

The programming syntax of a tuple is exactly the same as its mathematical counterpart presented in Section 1.1.3, meaning that the tuple components are listed within a pair of parentheses and each component is separated by a comma. For example, the tuple

```
(-1.3,5,false,"sam",{1,4,5},'a')
```

has a length equal to six, whose components are the real number -1.3, the integer 5, the Boolean value `false`, the string "sam", the set $\{1, 4, 5\}$, and the character 'a'. Since a tuple can have several different types of components, for the sake of redability it's sometimes helpful to give each one a name, as in

```
(v = -1.3,freq = 5,repeats = false, name = "sam",
    components = {1,4,5},file_code = 'a')
```

the names on the left of each option must follow one of the variable-naming conventions (upper or lower) described in Section 2.5. Also, when writing a tuple you must either name each component, or leave each component unnamed. There is no middle ground. For example, the tuple

```
(x = -2, y = 7)
```

has two components whose respective values are -2 and 7, while the tuple

```
(x = -2, 7)
```

has two components whose respective values are $x = -2$ and 7, meaning that the first component is *not* equal to -2, but rather equals the option $x = -2$. Indeed,

```
> (x = -2) == -2

false
```

evaluates to false since an option is not the same thing as an integer.

3.1.1 Structure attributes

Associated with every structure is one or more attributes. For example, associated with a tuple t is its `length` attribute that equals t's number of components. Moreover, associated with each attribute is a postfix function for accessing its value. For example, the postfix function `.length` takes as input a tuple t and returns the length of the tuple. Its signature is thus

`.length(Tuple,Integer)`,

where `Tuple` represents the set of all tuples, and `Integer` is the set of integers. For example,

```
> (1,4,5,7).length

4
```

evaluates to 4, since input tuple $(1, 4, 5, 7)$ has four components. In general, the postfix function associated with an attribute shares the same name as the attribute, along with a "dot" that separates the input from the attribute name.

3.1.2 Accessing components

The purpose of a structure is to bring together entities that together form a larger aggregate entity. For example, a steering wheel, tires, brakes, chassis, engine, etc. go together to form an automobile. This brings up the problem of accessing the constituent parts of an aggregate. In programming we resolve this problem by first assigning each constituent an address. In the case of a tuple having length n, its component constituents are assigned the respective addresses $0, 1, \ldots, n-1$.

Example 3.1.1. Given the tuple

```
(-1.3,5,false,"sam",{1,4,5},'a')
```

The following table gives the address for each of its components.

Component	-1.3	5	`false`	"sam"	$\{1, 4, 5\}$	'a'
Address	0	1	2	3	4	5

□

In general, the i th component of a tuple has address $i-1$. Now that each component has an assigned address, we may access it by placing its address, delimited by a pair of square brackets, next to the tuple. For example, to access the third component of

$$(10, -9, 8, -7, 6),$$

we know its address is 2, and so we have the following.

```
> (10,-9,8,-7,6)[2]

8
```

Alternatively, we could first assign the above tuple to a variable, and then apply the address to the variable as follows.

```
> t := (10,-9,8,-7,6)

success

> t[2]

8
```

3.1.3 Strings as tuples

It's worth mentioning that a string is considered a special kind of tuple for which each component consists of a character. The only difference is that a string may be represented as a sequence of characters delimited by double quotes. For example, we have the following.

```
> ['h','e','l','l','o']

hello
```

```
> "hello"

hello

> "hello".length

5

> "hello"[4]

o
```

3.1.4 The Prod function

One way of defining a list of tuples is to use list notation, as in

$$\{(1,2,-5),(3,1,4,5,-3),(),(-4,1,4)\}.$$

In this section we introduce the `Prod` function whose input sets are $A_1, A_2, \ldots, A_n$, and whose output is the Cartesian product $A_1 \times A_2 \times \cdots \times A_n$. For example,

```
> Prod(1..3,Boolean,toset "abcd")

Prod(1..3,Boolean,{'a','b','c','d'})
```

defines the Cartesian product of tuples having length three, and whose respective components are integers between 1 and 3, Boolean values, and characters between ‘a’ and ‘d’.

The following are some useful facts regarding `Prod`.

1. `Prod` allows for positive-integer inputs that represent the *duplicity* of the previous set input A, i.e. the number of subsequent components having domain set equal to A. For example,

   $$\texttt{Prod}(\texttt{Boolean}, 3, \texttt{Real})$$

   evaluates to the Cartesian product of four sets, the first three of which are equal to `Boolean`, while the fourth is equal to `Real`.

2. If only one set A is provided as input to `Prod`, then the output set is the set of tuples of *all* possible lengths, where the domain of each component is equal to A. This set may be mathematically described as

   $$\bigcup_{i=0}^{\infty} A^i,$$

   where $A^0 = \{(\ )\}$, and

   $$A^i = \underbrace{A \times A \times \cdots \times A}_{i \text{ times}}.$$

   Note that for a one-dimensional Cartesian product over A, we use $\texttt{Prod}(A, 1)$.

3. `Prod` has the optional input `lengths` that allows for an input that is a set of additional lengths that may be assumed by some members of the output set. For example,

   $$\texttt{Prod}(A, 6, \text{lengths} = \{0, 2, 4\})$$

   outputs the set $A^0 \cup A^2 \cup A^4 \cup A^6$, which is the union of three Cartesian products along with the empty tuple having length 0. This is especially useful for sets of strings having different lengths.

4. `Prod` also has the optional input `labels` which is a tuple of identifier labels, one for each component of the output product. For example,

   $$\texttt{Prod}(\{1, 2, 3, 4\}, 3, \text{labels} = (\text{length}, \text{width}, \text{height}))$$

outputs the Cartesian product D^3 which includes tuples such as

$$(\text{length} = 2, \text{width} = 2, \text{height} = 1).$$

Bibliography

[1] A. Allain *Jumping into C++*. Cprogramming.com, 2013

[2] H. Anton *Elementary Linear Algebra, 10th ed.* Wiley Publishing, 2010

[3] B. Bush *Solving the Shirokuro puzzle constraint satisfaction problem with backtracking: a theoretical foundation* Masters Thesis, Cal. State University, Los Angeles, 2012

[4] W. Cook *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation.* Princeton University Press, 2014

[5] N. Eén, N. Sörensson *"An Extensible SAT-solver"*, *SAT* 2003. http://minisat.se/downloads/MiniSat.pdf

[6] B. Kernighan and D. Ritchie *The C Programming Language, 2nd ed.*. Prentice Hall, 1988

[7] C. Larman *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed.* Prentice Hall, 2004

[8] K. Marriot, P. Stuckey *A Minizinc Tutorial*

`http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf`

[9] K. Rosen *Discrete Mathematics and Its Applications.* McGraw-Hill Education, 2011

[10] C. Sinz *Practical Applications of SAT* Invited Talk, Research Institute for Symbolic Computation (RISC), Linz, Austria, 2005, http://www.carstensinz.de/talks/RISC-2005.pdf

[11] R. Sosic, J. Gu *A Polynomial Time Algorithm for the N-Queens Problem* Sigart Bulletin, Vol. 1, 3, pp. 7-11, Oct. 1990

[12] F. Rossi, P. Van Beek, T. Walsh, Editors *Handbook of Constraint Programming.* Elsevier Science, 2006

[13] R. Stoll *Set Theory and Logic.* Dover Publications, 2012

[14] H. Schildt *Java: A Beginners Guide, 6th ed..* Mcgraw-Hill Osborne Media, 2014

[15] *Test Your Logic.* Dover Publications, 1972

[16] J. Warmer, A. Kleppe *The Object Constraint Language: Precise Modeling with UML* Addison-Wesley, 1998

[17] R. Weeks *Evaluating Expressions.* Amazon Digital Services, 2012

[18] J. Zelle *Python Programming: an Introduction to Computer Science, 2nd Ed..* Franklin, Beedle and Associates Inc., 2010