

CSC 202 Mathematics for Computer Science
Lecture Notes

Marcus Schaefer
DePaul University¹

Chapter 3

Databases and First Order Logic

We have seen close connections between relational databases and two logical theories: propositional logic and set theory, but both of them fall short of capturing relational databases completely; for example, they cannot even define the notion of a key formally. For that we need a more powerful tool: first-order logic.

Propositional logic is the logic of propositions, statements that are either true or false. Propositional logic does not care what the statements are about. Indeed, when we applied propositional logic to explain the logic of SQL's `WHERE` clause we already added the caveat that a condition like `Career = 'UGRD'` is not a proposition. Only after replacing `Career` by a value in a particular field does the condition become true or false. Set theory was different in that it dealt with collections of entities, whose nature was of no interest to us other than that they belonged to some sets and not to others. Behind set theory lurks first-order logic which allows us to express relationships between the members of our universe.

In this section we start with the mathematical point of view. Don't think about databases for a while.

3.1 Relations

When we say something like “Lisa is a sister of Bart” or “Homer is the father of Maggie” we are expressing a relation between two different objects (Simpsons in this case). In mathematics your objects might be natural numbers, graphs or propositions, but the principle is still the same: when writing $3 < 4$ we are expressing a relation between 3 and 4. We are just working in a different universe, but there are more similarities than differences. For example, if somebody is a brother to somebody else, and that somebody else is a brother to yet another person, then the first person is a brother to the last person. Sound cumbersome?

That's why we have mathematics.

When a twelfth-century youth fell in love he did not take three paces backward, gaze into her eyes, and tell her she was too beautiful to live. He said he would step outside and see about it. And if, when he got out, he met a man and broke his head—the other man's head, I mean—then that proved that his—the first fellow's—girl was a pretty girl. But if the other fellow broke his head—not his own, you know, but the other fellow's—the other fellow to the second fellow, that is, because of course the other fellow would only be the other fellow to him, not the first fellow who—well, if he broke his head, then his girl—not the other fellow's, but the fellow who was the— Look here, if A broke B's head, then A's girl was a pretty girl; but if B broke A's head, then A's girl wasn't a pretty girl, but B's girl was. That was their method of conducting art criticism.

Jerome K. Jerome. *Idle Thoughts of an Idle Fellow*.

We write xRy to express that x is in relation R to y . So if R is the relation “is brother of”, then xRy expresses that x a brother of y ; with this we can express the idea that a brother of a brother is still a brother as: xRy and yRz implies xRz , or even shorter:

$$xRy \wedge yRz \rightarrow xRz.$$

Note that the same is true if R is the relation “is smaller than” and the universe are the natural numbers (or the real numbers). The relation R is *transitive* in both cases. It turns out to be useful to study properties of relations such as transitivity in general without limiting ourselves to a particular universe. (Though it is always a good idea to think of specific examples.)

The relation “is brother of” is *binary*: it relates *two* elements of the universe. Relations can be more complex, for example “ x places an order for y at time t ” is a *ternary* relation in that it relates three objects (this universe obviously includes people, time and shopping goods). Charles Peirce was fond of the example “ x is a giver of y to z ”, another ternary relation; similar to it is the supplier relationship often found in databases: “company x supplies product y to customer z ”. Ternary relations cannot naturally be written like binary relations with the relation symbol appearing between the objects it relates, like $3 < 4$ or $3 + 4 = 7$. So we move to another notation: we write $R(x, y)$ or $R(x, y, z)$ denote that R is true for the pair (x, y) or the triple (x, y, z) (and similarly for any other tuple). The number of objects in the relation is the *arity* of the relation. There is one more important special case, *unary* relations that have just one object, written $R(x)$. Unary relations are also known as *predicates* or *attributes* (note that we are sweeping tons of philosophical literature under the carpet by claiming that). An earlier example we saw would be “is human” which we would now write as “human(x)”; other examples include “is orange” and “is even”.

Relations, well, relate different objects in our world; they are not just what makes relational databases, but they are central to mathematics and any science. Not surprisingly, then, relations are well-investigated and we have come to distinguish between many different types of relations. There are ordering relations (“being taller than”, “being less than”) and equivalence relations (“being in the same class as”, “being of the same sex”). To capture the essential nature of these types, we first need to abstract some properties of relations.

A binary relation R is *reflexive* if $R(x, x)$ for all x . For example, \leq is reflexive, but $<$ is not.

A binary relation R is *symmetric* if $R(x, y) \rightarrow R(y, x)$ for all x and y . For example, $=$ is symmetric, but \leq is not.

A binary relation R is *transitive* if $R(x, y) \wedge R(y, z) \rightarrow R(x, z)$ for all x , y and z . For example \leq is transitive, but “being a friend of” is not (although it gets claimed repeatedly).

Example 3.1.1. Let us try some examples. Start with $R(x, y) = “x \text{ likes } y”$ (our universe being humanity). There are certainly x for which $R(x, x)$ is true, that is, humans that like themselves, but we don’t have $R(x, x)$ for all x (that is, there are humans that do not like themselves), so R fails to be reflexive. Similarly, it is not symmetric, since x liking y does not always imply that y likes x , and it is not transitive, since x can dislike z , even if z is liked by y and y is liked by x .

What about $R(x, y) = “|x - y| \text{ is even}”$ (our universe being the natural numbers). $|x - x| = 0$, which is even, so R is reflexive. Also, since $|x - y| = |y - x|$, $R(x, y)$ is symmetric. Finally, if there is an even difference between x and y and an even difference between y and z , then there is an even difference between x and z , so this relation is transitive as well.

Exercise 3.1.2. For a more mathematical example of non-transitivity, consider being coprime. Two numbers are *coprime* if their greatest common divisor is 1. Show that being coprime is not transitive.

Exercise 3.1.3. For each of the following relations determine whether it is (a) reflexive, (b) symmetric, and (c) transitive. For each property argue why the relation has the property, or give a counterexample if it fails to have the property.

1. (S) “is sibling of”,
2. “has the same parents as”,
3. “is enrolled in one of the same courses as”,
4. “is enrolled in all the same courses as”,
5. “is coprime to”,
6. “is brother of”,

7. “is father of”,
8. “is cousin of”,
9. “is older than”,
10. “has won a game of tennis against”.

With these notions we can already define the most important class of relations: a binary relation is an *equivalence relation* if it is reflexive, symmetric and transitive. An equivalence relation R naturally groups objects: any x is related by R to itself, and if it is related by R to y , then y is also related by R to x , and if x is related to y by R and y to z , then x is also related by R to z . Exactly the behavior you would want if you needed to classify objects by being in the relation R to each other (think, for example, about objects sharing the same color). Let R be an equivalence relation, and for every element a define

$$[a]_R := \{b : R(a, b)\},$$

the *equivalence class* or a under R , namely all those elements that are equivalent to a under R (e.g. all objects having the same color as a). Since R is an equivalence relation, then any two elements in $[a]_R$ are related by R : pick $x \in [a]_R$ and $y \in [a]_R$. That is $R(a, x)$ and $R(a, y)$. But then $R(x, a)$ (symmetry), and, therefore, $R(x, y)$ (transitivity). We call $[a]_R$ the *equivalence class* of a . It contains everything *equivalent* to a and, as we just saw, everything a is equivalent to, is equivalent to each other as well, so we are creating a category.

Example 3.1.4. Consider $R =$ “has the same sex as”. Pick any person a , then $[a]_R$ will contain all people of the same sex as a . In particular, any two people within $[a]_R$ have the same sex. So, the notion of sex could be defined from a (scientifically possibly more primitive) notion of “having the same sex”. On the Simpsons, R leads to two equivalence classes: $\{\text{Maggie, Marge, Lisa}\}$ and $\{\text{Homer, Bart}\}$. So, for example, $[\text{Homer}]_R = \{\text{Homer, Bart}\} = [\text{Bart}]_R$.

On the other hand, if we let $R(x, y) =$ “ x has the same parents as y ”, the Simpsons split into three equivalence classes: $[\text{Homer}]_R = \{\text{Homer}\}$, $[\text{Marge}]_R = \{\text{Marge}\}$, and $[\text{Lisa}]_R = [\text{Bart}]_R = [\text{Maggie}]_R = \{\text{Lisa, Bart, Maggie}\}$. Note that an equivalence class is never empty, it always contains at least one element.

The most prominent example of an equivalence relation is equality, of course, and that is what the definition of an equivalence relation is modeled on. It is more general, of course, for example, it can capture equality of objects in a certain respect.

Example 3.1.5. For $R =$ “has the same age as”, the equivalence classes $[a]_R$ will consist of all the different age groups. We are grouping people with respect to their age. (R would split the Simpsons into 5 different equivalence classes, since everybody has a different age.)

Example 3.1.6. We now see clearly the difference between “is a sibling of” and “has the same parents as”. While “is a sibling of” is symmetric, it is not transitive, and, while we can define the set $[a]_{\text{is a sibling of}}$ of all the siblings of a , this set could contain two people who are not siblings of each other. On the other hand “has the same parent as” is reflexive, symmetric and transitive, so any two people in $[a]_{\text{has the same parents as}}$ will have the same parents, and this is a possible way of classifying people.

Example 3.1.7. Let us return to an earlier example: $R(x, y) = “|x - y|$ is even” over the universe of natural numbers. As we saw before, this R is reflexive, symmetric and transitive, so it is an equivalence relation. What are the equivalence classes it defines. $[0]_R$ contains all natural numbers that have an even difference with 0, in other words, all even numbers. On the other hand, $[1]_R$ contains all natural numbers that have an even difference from 1, which is the set of all odd numbers.

$$\begin{aligned} [0]_R &= \{0, 2, 4, 6, 8, \dots\} \\ [1]_R &= \{1, 3, 5, 7, 9, \dots\} \end{aligned}$$

And these are all the equivalence classes of R over the natural numbers there can be, since $\{0, 2, 4, 6, 8, \dots\}$ and $\{1, 3, 5, 7, 9, \dots\}$ together contain all natural numbers. If we looked at

$$[3]_R$$

for example, we find that $[3]_R = \{1, 3, 5, 7, 9, \dots\} = [1]_R$, since $3 \in [1]_R$. To represent the equivalence classes of the natural numbers with respect to R we could choose $[16]_R$ and $[43]_R$, or course, but $[0]_R$ and $[1]_R$ are more canonical (they are smaller, for one). So 0 and 1 are the *standard representatives* of the natural numbers after applying the equivalence relation R .

If we rephrase this in plain English, all we have just said is that natural numbers come in two types: even and odd, and we can easily determine whether two numbers are of the same type by calculating their difference and making sure it is even. This particular way of looking at the natural numbers is quite important; we will encounter it again, as *parity* when talking about games and puzzles, and as part of modular arithmetic when talking about cryptography.

Exercise 3.1.8. Consider the relationship $R(x, y) = “|x - y|$ is a multiple of 3”.

- Show that R is an equivalence relation.
- Find the equivalence classes of R .

Excursion: Databases and Grouping

In a sense, SQL allows you to build a very limited type of equivalence classes by allowing you to *group* records by values of fields. For example:

```
SELECT City, count(*)
FROM Student
GROUP BY City;
```

will group the records in the student table by the value of the field `City`. Adding `count(*)` to the `SELECT` clause gives us a count of the number of elements in each group. We can also group by multiple attributes; for example, if we wanted to know what degrees students are in, grouping just by `Career` or `Program` will not give us the relevant information (try it). But we can group by both simultaneously:

```
SELECT Career, Program, count(*)
FROM Student
GROUP BY Career, Program;
```

This works with multiple tables and conditions as well.

```
SELECT Program, min(started)
FROM Student
WHERE Career = 'UGRD'
GROUP BY Program;
```

tells us the first year a particular student started in a program (by program).

Remark 3.1.9. Two important remarks about how `GROUP BY` works: the `WHERE` is checked first—that is, the table is reduced to the records fulfilling the `WHERE` condition—and *then* the records are grouped by the attributes listed in the `GROUP BY` clause. At that point, counting or any other type of accumulation is done (you can sum numbers using `sum`, or compute their maximum or minimum or average values, using `max`, `min` and `avg`).

Secondly, if you are selecting attributes in the `SELECT` clause of a grouped SQL query, you can only select attributes that have explicitly been grouped by. For example, if we want to count how many students have taken each course, we might be inclined to write:

```
SELECT CID, Department, CourseName, count(*)
FROM Course, Enrolled
WHERE CourseID = CID
GROUP BY CID;
```

This makes sense, since for each `CID` there is a unique `Department` and `CourseName`. But how would your database engine know that this is the case? This is why the SQL standard would force you to write

```
SELECT CID, Department, CourseName, count(*)
FROM Course, Enrolled
WHERE CourseID = CID
GROUP BY CID, Department, CourseName;
```


H2 and some other systems will allow the first form of the query; we will be more conservative and stick to the SQL standard and explicitly group by any attribute we need to select.

- Exercise 3.1.10.**
1. (S) Write a query that lists the first and last year for each program that a student started in it.
 2. By career, find the average date that the students in that career started.
 3. Find the total number of students that ever enrolled in a class (i.e. total over all quarters and years).
 4. Find the total number of students in every class taught (that is, by quarter and year as well).

Example 3.1.11. Incidentally, you can use the result of an accumulation in another query, e.g.

```
SELECT *
FROM student
WHERE started = (SELECT min(started)
                 FROM Student);
```

This gives you the first cohort of students at your university.

- Exercise 3.1.12.**
1. Find the president of the oldest student group.
 2. For each student group, list its senior members (i.e. current members that joined earlier than other current members).

How would we find student groups that have at least ten members? We know how to get the counts for each student group:

```
SELECT Name, count(*)
FROM StudentGroup, MemberOf
WHERE GroupName = Name
GROUP BY Name;
```

We want to restrict this list to those entries where the count is at least ten. We cannot do this in the `WHERE` clause, since that clause gets processed *before* the accumulation, so no accumulated information is available at that point. We have to add this condition *after* the grouping. For this, there is an extra clause, called `HAVING` that will be run on the output of the grouping:

```
SELECT Name, count(*)
FROM StudentGroup, MemberOf
WHERE GroupName = Name
GROUP BY Name
HAVING count(*) >= 10;
```

- Exercise 3.1.13.**
1. (S) Find student groups that have no members.

2. Find student groups that have one or two members.
3. Find students that have joined more than one student group.
4. List years in which at least two students started at the university.
5. Find students that are enrolled full-time (four classes or more).
6. Find classes that have had an average enrollment of at least ten students.

To deepen our understanding of equivalence classes, let us develop an alternative way of looking at them, already suggested by our earlier examples. Suppose we have split our universe U into several sets (or categories, think of age, sex, etc.) that are pairwise disjoint. More formally, we call a collection U_i , where $i \in I$ a *partition* of U if

$$U = \bigcup_{i \in I} U_i$$

and $U_i \cap U_j = \emptyset$ for all $i \neq j$, $i, j \in I$. If we have a partition of U we can define a relation R on U as follows: $R(x, y)$ is true if and only if x and y belong to the same U_i , $i \in I$.

Exercise 3.1.14. Show that the R defined in this way is an equivalence relation: verify reflexivity, symmetry and transitivity.

On the other hand, suppose we are given an equivalence relation R over the universe U . Look at the collection of all equivalence classes: $U_a = [a]_R$, for all $a \in U$. First of all note that

$$U = \bigcup_{a \in I} [a]_R$$

since every $a \in U$ is contained in some equivalence class, namely U_a (since R is reflexive). However, the U_a are not a partition of the universe U , since two of them could overlap. However, suppose U_a and U_b overlap: that is, there is a c such that $c \in U_a$ and $c \in U_b$, or, in other words, $R(a, c)$ and $R(b, c)$. Since R is symmetric and transitive, we can conclude that $R(a, b)$, so $b \in U_a$, and, therefore $U_b \subseteq U_a$, and, using the same argument with a and b exchanged, $U_a \subseteq U_b$, and, thus $U_a = U_b$. That is, if two equivalence classes overlap, they are identical. So we can simply drop all indices in U that lead to duplication, to get a set $I \subseteq U$ such that $a, b \in I$ implies that $U_a \neq U_b$, and, therefore, $U_a \cap U_b = \emptyset$. In other words, U_a with $a \in I$ is a partition of U .

In summary: you should be thinking of equivalence relations as a partition of the universe into categories.

We already mentioned another useful class of relations, ordering relations, such as \leq . If we go back, it seems that ordering relations should be reflexive

and transitive, but certainly not symmetric. If $x \leq y$ and $y \leq x$, indeed, we would like to conclude that $x = y$. This is the missing property we need to define an ordering relation:

A binary relation R is *anti-symmetric* if $R(x, y) \wedge R(y, x) \rightarrow x = y$.

An *ordering relation* is a reflexive, anti-symmetric, transitive relation.

Example 3.1.15. We can order the Simpsons by age: Homer $>$ Marge $>$ Bart $>$ Lisa $>$ Maggie (Homer's age fluctuates a bit). If we order them by how tall they are, we get Marge $>$ Homer $>$ Bart $>$ Lisa $>$ Maggie, so the same set can support many different orderings.

Exercise 3.1.16. Show that $x|y$ (“ x divides y ”) is an ordering relation on the natural numbers. Is $x|y$ an ordering relation on the integers? On the real numbers?

Exercise 3.1.17. Show that \subseteq is an ordering relation on sets.

The previous examples shows that ordering relations can be quite different from our original model, \leq . The main difference is that two elements might not be *comparable* by the relation. For numbers x and y we always have $x \leq y$ or $y \leq x$, but it is not true that for an arbitrary ordering \preceq we always have either $x \preceq y$ or $y \preceq x$. Indeed, $\{\text{Marge}\} \not\subseteq \{\text{Homer}\}$ and $\{\text{Homer}\} \not\subseteq \{\text{Marge}\}$. So while, as we saw, \subseteq is an ordering relation to our definition, the two sets $\{\text{Marge}\}$ and $\{\text{Homer}\}$ are not even comparable.

An ordering relation \preceq for which either $x \preceq y$ or $y \preceq x$ for all elements x and y is called *total*. Otherwise it is called *partial*.

Example 3.1.18. Consider the set W of all possible words (arbitrary sequences of letters, including “szyzgy” but also “sdhfsjdf”). This set can be totally ordered as follows: given two words $x \neq y \in W$, let w be their longest common prefix.

If $w = x$ or $w = y$, then let $x \leq_{\text{lex}} y$ if $|x| \leq |y|$, where $|x|$ and $|y|$ are the lengths of x and y . If neither $w = x$ nor $w = y$, then both have a letter following w , that is, $x = w\ell_1 \dots$ and $y = w\ell_2 \dots$, where ℓ_1 and ℓ_2 are letters of the alphabet. We let $x \leq_{\text{lex}} y$ if ℓ_1 comes before ℓ_2 in the alphabet. The resulting ordering is called the *lexicographic* or *dictionary* ordering of words (there are other ways to order words). It is a simplified variant of the ordering used by dictionaries to arrange words.

For example, $x = \text{“hand”}$ and $y = \text{“handle”}$ have the longest common prefix $w = \text{“hand”}$, which coincides with x , so we let $\text{“hand”} < \text{“handle”}$, since “hand” is shorter. For $x = \text{“brutal”}$ and $y = \text{“brunch”}$, we have a longest common prefix of $w = \text{“bru”}$. In x , “bru” is followed by “t” whereas in y “bru” is followed by “n” . Since $\text{“n”} \prec \text{“t”}$ in the English ordering of the alphabet, we let $\text{“brunch”} < \text{“brutal”}$.

The lexicographic ordering does disagree with our standard ordering of the numbers (if we extend the notion of words to be made from letters as well as numbers). If we considered 112 and 12 as words “112” and “12”, then “112” would be listed before “12”, while, as numbers, $12 < 112$.

We did not specify which alphabet we are using, and the lexicographic ordering of words makes sense for any alphabet, be it the English alphabet, or the binary alphabet $\{0, 1\}$, or even an infinite set. Even more, we can start with any ordering of the alphabet we like. For a dictionary, we would order the English alphabet as “a” < “b” and so on. For a spam filter, where we would want to check for more frequent words first, we could order the letters in order of frequency, so “e” ; “t” ; “a” and so on, since “e” is the most frequent letter in English, “t” the second-most frequent, and so on.

Exercise 3.1.19. Show that the lexicographic ordering $x \leq_{\text{lex}} y$ is indeed a total ordering relation.

Exercise 3.1.20. Show that the lexicographic ordering $x \leq_{\text{lex}} y$ is not *well-founded*, in the sense that you can build an infinite sequence of words w_1, w_2, \dots such that $w_1 \geq_{\text{lex}} w_2 \geq_{\text{lex}} w_3 \geq_{\text{lex}} \dots$, where $x \geq_{\text{lex}} y$ if $y \leq_{\text{lex}} x$. *Note:* This shows that the lexicographic ordering is not very useful for certain approaches to algorithmic processing, where you would want each element to have only a finite number of predecessors in the order, like the natural numbers.

For numbers we distinguish between < and \leq and the same distinction can be made for arbitrary orders. We call \prec a *strict ordering relation* if it is anti-reflexive, anti-symmetric and transitive, where a binary relation R is called *anti-reflexive* if $\neg R(x, x)$ for all x . As we did for the non-strict orderings we distinguish between partial and total strict orders. A strict ordering \prec is *total* if for any two distinct elements x and y we either have $x \prec y$ or $y \prec x$. Note that we cannot require this for all pairs of elements, since for $x = y$ we do not have either $x \prec y$ or $y \prec x$.

Note that by definition strict ordering relations are not ordering relations (since they are not reflexive, indeed the opposite).

Excursion: Databases and Ordering

SQL allows you to sort your output by any field; for example,

```
SELECT *
FROM Student
ORDER BY Started;
```

will list the students in the order that they started in. If we wanted to sort the students within each year by last name and then by first name, we could do so by using

```
SELECT *
FROM Student
ORDER BY Started, LastName, FirstName;
```

In passing, we already hinted at some standard operations on relations. For example, we saw that with the lexicographic ordering \leq_{lex} automatically came a second lexicographic ordering \geq_{lex} : one ordering puts the smallest elements first, the other puts the larger elements first. For example, if we order 1, 2, 3 by $<$ we get $1 < 2 < 3$. If we order it by $>$ we get $3 > 2 > 1$. Both are perfectly fine orders. In general, starting with a binary relation $R(x, y)$ we can always consider the *inverse relation* $R^{-1}(x, y) := R(y, x)$. So $<^{-1} = >$. This construction also works for binary relations that are not orders.

Example 3.1.21. If $R(x, y) =$ “ x is parent of y ”, then $R^{-1}(x, y)$ is “ x is child of y ”.

Exercise 3.1.22. What are the inverses of the following relations:

1. “is ancestor of”,
2. “is brother of” (careful),
3. “is uncle of”,
4. “is coprime to”,
5. “has the same parents as”,
6. “is older than” (careful),
7. “has won a game of tennis against”.

Exercise 3.1.23. Show that R is symmetric if and only if $R^{-1} = R$.

Another natural operation on relations is to combine them. In the easiest case, you just look at the relation “once removed”. A parent of a parent is a grandparent. A child of a child is a grandchild. Or combine them in arbitrary connections: a child of a sibling, a sibling of a parent, and so on.

More formally, let $R(x, y)$ and $S(u, v)$ be two relations. The *join* of R and S , written $R \circ S$ is the relation that holds between x and v if there is a y such that $R(x, y)$ and $S(y, v)$.¹

Example 3.1.24. Let us go back to the parent example: if $R(x, y) =$ “ x is parent of y ” and $S(u, v)$ is the same relation, that is, $S(u, v) =$ “ u is parent of v ”, then $R \circ S(x, v)$ is true if there is a y such that x is parent of y and y is parent of v . In other words, it is true if x is a grandparent of v .

Let us try another example. Let $R(x, y) =$ “ x is sibling of y ” and $S(u, v) =$ “ u is parent of v ”. Then $R \circ S(x, v)$ is true if there is a y such that x is a sibling of y and y is a parent of v . In other words, $R \circ S(x, v)$ means that x is uncle or aunt of v (excluding uncles or aunts by marriage).

¹In database theory or relational algebra, the join is also known as the *equijoin* since you join on a condition of equality; it is a special case of the Θ -*join*.

Example 3.1.25. Suppose $R(x, y) = “x = 2y”$ and $S(u, v) = “u = 3v”$. Then $R \circ S(x, v)$ is true if there is a y such that $x = 2y$ and $y = 3v$. Or, in other words, if $x = 2(3v) = 6v$.

Exercise 3.1.26. What is the join $R \circ S$ in the following examples:

1. $R(x, y) = “x$ is married to $y”$ and $S(u, v) = “u$ is a child of $v”$,
2. $R(x, y) = “x$ is a child of $y”$ and $S(u, v) = “u$ is married to $v”$,

The join of a relation with itself is closely related to its transitivity: “parent of parent” is “grandparent” and not the same as “parent”; however, “taller than somebody taller than” is the same as “taller than”. And being a parent is not transitive, while being taller than somebody else is. We will formalize this relation (and several others) in the next section.

Joins can be defined on arbitrary, not just binary, relations. In that case, we need to specify which argument of R is matched up with which argument of S . At this point, you probably realize that we have seen, and indeed, used joins before: when connecting multiple tables in a database. We join them by requiring that the foreign key be the same as the primary key. Consider the following query listing the last names of presidents of student groups:

```
SELECT LastName, SID, Name
FROM Student, StudentGroup
WHERE PresidentID = SID;
```

This is a join of `Student(ln, fn, sid, ssn, cr, pr, ct, st)` and `StudentGroup(pid, gn, fd)`, joined by the condition that $sid = pid$; i.e. we are looking at the relation we obtain from combining each record from `Student` with `StudentGroup` and restricting to those records where $sid = pid$. This corresponds exactly to the definition of a join (except that so far we’ve always implicitly joined the last attribute of the first relation to the first attribute of the second relation; the current join is a bit more general in that we can specify which attributes get joined). And, indeed, many database systems support a JOIN operation directly, that is, we could write the query for student group presidents as:

```
SELECT LastName, SID, Name
FROM Student JOIN StudentGroup ON PresidentID = SID;
```

This allows for a clean separation of the foreign key/primary key requirements and more specialized query restrictions, e.g. when we write

```
SELECT LastName, SID, Name
FROM Student JOIN StudentGroup ON PresidentID = SID
WHERE founded > 2000;
```

for all presidents of student groups founded after 2000. However, some systems do not support the JOIN keyword, which is why we will not use it explicitly.

3.2 The Extensional View

Our examples of relations in this chapter so far have included many natural relations, like “sibling”, “parent”, $<$ and so on. These relations seem to differ inherently from the type of relations we encounter in a database though, say a student enrolled in a course. A relationship like “sibling” or $<$ is defined, through words or mathematics, such relations are called *intensional*,² since they are determined by their intension, their meaning. In a database which students take a class is explicitly listed, such relations are called *extensional*, since they are determined by their extension. Here we have one of the basic tensions in computer science, between the intensional and the extensional, in a nutshell. Whenever you represent information in a computer, you have to make a decision about whether the presentation will be intensional or extensional: do you work with the meaning or with the explicit cases? Much of artificial intelligence tends towards intensional representations, which means artificial intelligence needs good implementations of logic to process semantical rules. At the other end of the spectrum we have relational databases, where all relations are given explicitly. If we want to store who is married to whom, we need to do so explicitly, and tell the database that William Shakespeare was married to Anne Hathaway. But there is no way to tell the database that marriage is a symmetric relation and that this implies that Anne Hathaway was married to Shakespeare. If we need to use that information as well, we need to store it, or work the symmetry into our queries.³

Actually, we have discussed the extensional view before, when talking about pairs and tuples: we can view a table in a database as a subset of a Cartesian product over its domains; we call this the *set-theoretic* view of relations. The example we saw was `studentgroup` which we explained as a subset of $A \times B \times C$, where A is the set of all strings of length at most 40, B the set of integers with at most 5 digits, and C the set of all years. If we take the extensional view, we could also write $(x, y, z) \in \text{studentgroup}$ to talk about a particular record in the table `studentgroup`, rather than `studentgroup(x, y, z)`, which suggests a relation `studentgroup` that holds for (x, y, z) if x is the name of a student group founded in the year z and whose president is the student with student ID y .

Just as we allowed unions, intersections, and differences (complements) of sets and tables, we can do the same for relations by simply viewing them as sets.

Example 3.2.1. Let $R(x, y) = “x < y”$, and $S(x, y) = “x = y”$. Then $R \cup S(x, y) = “x \leq y”$. In this case $R \cap S(x, y) = \{\}$, since there are no x and y such that $x < y$ and $x = y$.

For another example, let $R(x, y) = “x \text{ sister of } y”$, $S(x, y) = “x \text{ is older than } y”$. Then $R \cup S(x, y) = “x \text{ is older than } y \text{ or a sister of } y”$. $R \cap S(x, y) = “x$ is

²Intensional. Not intentional.

³And we do not even want to touch on the philosophical issues involved in the distinction between extension and intension.

older sister of y ". Finally, $R - S(x, y) = "x$ is younger (or same-age) sister of $y"$.

Note that the set-theoretic operations immediately correspond to logical operations on the relation: $R \cup S$ is the same as $R \vee S(x, y) := R(x, y) \vee S(x, y)$, $R \cap S$ the same as $R \wedge S(x, y) := R(x, y) \wedge S(x, y)$, and $R - S$ the same as $R \wedge \overline{S}(x, y) := R(x, y) \wedge \overline{S}(x, y)$.

Exercise 3.2.2. You are given relations $R(x, y) = "x$ is a child of $y"$, $S(u, v) = "u$ is a sibling of $v"$, $T(w, z) = "w$ is married to $z"$. Set-theoretically express the relations x is aunt or uncle of y , and x is nephew or niece of y .

For binary relations there is a good alternative to the set-theoretic view; for example, picture the relation "is parent of" in the world of the Simpsons.

	Homer	Marge	Lisa	Bart	Maggie
Homer	\perp	\perp	\top	\top	\top
Marge	\perp	\perp	\top	\top	\top
Lisa	\perp	\perp	\perp	\perp	\perp
Bart	\perp	\perp	\perp	\perp	\perp
Maggie	\perp	\perp	\perp	\perp	\perp

Here is an example of how to read the matrix: Marge is a parent of Maggie as witnessed by the entry \top in the row labeled Marge and the column labeled Maggie. In one more step of abstraction, we remove the labels of rows and columns (we have to agree on what they mean, but that is typically clear), and replace \perp by 0 and \top by 1, and we get a 0/1-matrix, that is, a matrix all of whose entries are 0 or 1:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix, let's call it M , still encodes all the information we need to know about "is parent of" in the world of the Simpsons: $M[i, j]$, the entry is row i and column j is 1 if and only if the i th Simpson is a parent of the j th Simpson (where we order the Simpsons as Homer $<$ Marge $<$ Lisa $<$ Bart $<$ Maggie).

We call this the *matrix* view of binary relations.

Exercise 3.2.3. Construct matrices for the following relations on the Simpsons, ordered as Homer $<$ Marge $<$ Lisa $<$ Bart $<$ Maggie.

1. "is child of"; compare this to "is parent of". Do you see a connection?
2. "is older than",
3. "has same sex as",
4. "is identical to",

5. “is younger than”; compare this to “is older than”. Do you see a connection? What about the diagonal elements?

Let us go back to some of the properties we discussed earlier: being reflexive, symmetric, anti-symmetric, transitive, and so on. How do these translate into the extensional or the matrix view?

- A relation is reflexive, if the diagonal of the associated matrix contains only ones.
- A relation is symmetric, if the associated matrix remains unchanged when flipping it along the diagonal.

Flipping a matrix along the diagonal is called *transposing*: $M^T[i, j] := M[j, i]$. We claim that a relation R is symmetric if for its associated matrix M we have $M^T = M$. This is really just the same as saying that $R = R^{-1}$.

The matrix I consisting of ones in the diagonal, and zeroes everywhere else is known as the identity matrix. Formally, $I[i, j] = 1$ if $i = j$ and $I[i, j] = 0$ if $i \neq j$. We define addition of two matrices by component: $(A+B)[i, j] = A[i, j] + B[i, j]$.

Lemma 3.2.4. *A relation is reflexive, if its associated matrix M can be written as $M = I + M'$.*

Exercise 3.2.5. Give a characterization of anti-reflexive relations in terms of their associated matrix.

Exercise 3.2.6. Give a characterization of anti-symmetric relations in terms of their associated matrix. *Hint:* Subtraction of matrices is defined just like addition: $(A - B)[i, j] = A[i, j] - B[i, j]$. And there is a special matrix J with $J[i, j] = 1$ for all i, j which you might find helpful.

We have not characterized transitivity yet; it turns out that transitivity is closely related to the join of a binary relation with itself: “parent of parent” is “grandparent” and not the same as “parent”; however, “taller than somebody taller than” is the same as “taller than”. And being a parent is not transitive, while being taller than somebody else is. For the moment let us move back from the matrix point of view to the set-theoretic point view of relations.

Lemma 3.2.7. *$R \circ R \subseteq R$ if and only if R is transitive.*

Proof. Let us first show that $R \circ R \subseteq R$ implies that R is transitive. Arbitrarily pick $(x, y) \in R$ and $(y, z) \in R$. If we can show that $(x, z) \in R$, then we have shown that R is transitive, since (x, y) and (y, z) were chosen arbitrarily. Now, since $(x, y) \in R$ and $(y, z) \in R$, by definition of $R \circ R$, $(x, z) \in R \circ R$. But by assumption $R \circ R \subseteq R$, so $(x, z) \in R$, which is what we had to show.

For the other direction, let us assume that R is transitive. We need to prove that $R \circ R \subseteq R$. Pick an arbitrary $(x, z) \in R \circ R$. By definition of $R \circ R$ there has to be a y such that $(x, y) \in R$ and $(y, z) \in R$. But then, by transitivity of R , we can conclude that $(x, z) \in R$. Hence, we have shown that $R \circ R \subseteq R$. ■

Transitivity can also be expressed in the matrix view using matrix multiplication; this is a topic we will not follow up on here, but it is of practical interest, since it leads to the fastest algorithms known to compute the transitive closure of a relation.

The *transitive closure*? What is that? It is the answer to a very natural question: if a relation R is not transitive, can it be made so by adding connections to it? Take, for example, the relation “is parent of”. It is not transitive, since your grandmother is not your parent. But we could make it more transitive by also including grandparents. That still would not include the parents of grandparents. Well, include them as well. And so on. What we are doing is in each step replacing R with $R \cup R \circ R$, that is we add all (x, z) to the relation for which there is a one-step connection: $(x, y) \in R$ and $(y, z) \in R$. (Think about that step for $R =$ “is parent of”.) If we keep repeating this over and over, we will eventually cover the complete history of mankind, and R will correspond to the ancestor relationship. This process is called *transitive closure*, and the example here argues that the transitive closure of “is parent of” is the relation “is ancestor of”.

- Exercise 3.2.8.** 1. (S) What is the transitive closure of “is child of”?
2. For natural numbers, we define $S(n) = n + 1$, the *successor* of n . What is the transitive closure of “is the successor of”?

Computing transitive closures is important and time-consuming; to do so, we need the matrix point of view, a topic we will not go into more detail on here.

3.3 Quantification

We defined a relation R to be reflexive, if $R(x, x)$ for all x . Logically, there is something new here: we require the truth of a proposition for all objects x in the universe. This is beyond propositional logic, we have entered the realm of first-order logic. We used English to express for all x , but that can lead to confusion as our use of quantification gets more sophisticated, so we use a more mathematical notation:

$$(\forall x)[R(x, x)].$$

The parentheses are used mainly for readability, although they also tell us what part of the formula (enclosed by [and]), the quantifier applies to. In this example, there is no danger of being ambiguous, so we could also write $\forall x R(x, x)$.

The main new ingredient here is the quantifier \forall . More generally, if we have a relation R depending on x , we write

$$(\forall x)[R(x)]$$

to express that R holds whatever the value of x ; \forall is known as the *universal quantifier*. E.g. if $P(x) =$ “is American citizen” and $Q(x) =$ “has a social security

number”, then

$$(\forall x)[P(x) \rightarrow Q(x)]$$

expresses (whether true or not) that every American citizen has a social security number.

Exercise 3.3.1. What do the following formulas express, if, as above, $P(x)$ = “ x is American citizen” and $Q(x)$ = “ x has a social security number”.

1. (S) $(\forall x)[Q(x) \rightarrow P(x)]$,
2. $(\forall x)[P(x)] \rightarrow (\forall x)[Q(x)]$,
3. $(\forall x)[P(x)]$,
4. $(\forall x)[Q(x)]$,
5. $(\forall x)[P(x)] \wedge (\forall x)[Q(x)]$,
6. $(\forall x)[P(x)] \rightarrow (\forall x)[Q(x)]$ (again, yes),
7. $(\forall x)[Q(x) \leftrightarrow P(x)]$,
8. $(\forall x)[\overline{Q(x)}]$.

One conclusion we can draw is that the universal quantifier distributes over conjunction:

$$(\forall x)[P(x) \wedge Q(x)] \leftrightarrow (\forall x)[P(x)] \wedge (\forall x)[Q(x)].$$

The same is not true for disjunction, as the following example shows.

Exercise 3.3.2. Let $P(x)$ = “ x is male” and $Q(x)$ = “ x is female”. What do the following statements express?

1. $(\forall x)[P(x) \vee Q(x)]$,
2. $(\forall x)[P(x)] \vee (\forall x)[Q(x)]$.

Example 3.3.3. A *prime number* is a natural number larger than 1 whose only divisors are 1 and the number itself. This is a type of condition that is very naturally expressed using the universal quantifier: to express that n is prime, we need to say that any number dividing n is either 1 or n ; as earlier, we use $a|b$ to denote the relation “ a divides b ”. We can express that a divisor k of n has to be 1 or n using propositional logic:

$$k|n \rightarrow (k = 1 \vee k = n).$$

For n to be prime, every divisor of n has to be either 1 or n , so we need to require that any number that divides n has to be 1 or n :

$$(\forall k)[k|n \rightarrow (k = 1 \vee k = n)].$$

At this point we only need to add the requirement that n be larger than 1:

$$n \text{ is prime if and only if } n > 1 \wedge (\forall k)[k|n \rightarrow (k = 1 \vee k = n)].$$

Exercise 3.3.4. Using the predicate $a|b$ define “ n is a power of 2” without using exponentiation (2^b) (how do divisors of powers of 2 differ from divisors of other numbers). Can you define “ n is a power of 3”? How about “ n is a power of 4” ?

A formula can have multiple quantifiers, indeed, we have seen several examples of that already, albeit implicitly. What does

$$(\forall x)(\forall y)[R(x, y) \rightarrow R(y, x)]$$

express? How about

$$(\forall x)(\forall y)(\forall z)[R(x, y) \wedge R(y, z) \rightarrow R(x, z)]?$$

Exercise 3.3.5. 1. (S) Express the condition that the binary relation R is anti-reflexive.

2. Express the condition that the binary relation R is symmetric.
3. Express the condition that the binary relation R is anti-symmetric.
4. (S) Express the condition that the binary relation R is not reflexive.
5. Express that R is an ordering relation.
6. Express that R is an equivalence relation.

Exercise 3.3.6. Let $P(x, y) =$ “ x is taller than y ”. Express that if x is taller than y , then y cannot be taller than x .

Let $R(x, y) =$ “ x has social security number y ”. Using this relation let us try to express that everybody has *at most* one social security number. In other words, nobody can have two social security numbers. Suppose somebody did, that is, $R(x, y)$ and $R(x, z)$, then we would have to require that the two social security numbers are actually the same, viz. $y = z$. That was the missing piece, we can now write

$$(\forall x)(\forall y)(\forall z)[R(x, y) \wedge R(x, z) \rightarrow y = z].$$

Read this through again carefully, and interpret its meaning.

How would we express that somebody does have a social security number? Let $R(x, y) =$ “ x has social security number y ”, and suppose, for the moment, that x is one particular American. We want to say that there is a y such that $R(x, y)$. For that we use the *existential quantifier*:

$$(\exists y)[R(x, y)].$$

This, in itself, is a formula in the variable x , which we can quantify again, for example, to say that every American has a social security number:

$$(\forall x)(\exists y)[R(x, y)].$$

Again, reread this carefully, and then compare it to

$$(\exists y)(\forall x)[R(x, y)],$$

what is the difference in meaning? What about

$$(\exists x)(\forall y)[R(x, y)]?$$

We see that the order of quantifiers matters, we need to read them from the outside in (just as we do in SQL). Though often, to understand what the formula is saying, it helps to understand smaller piece of the formula first: recall how we built $(\forall x)(\exists y)[R(x, y)]$ in the first place: we built $(\exists y)[R(x, y)]$ first, and then added the universal quantifier. When reading the full formula, $(\forall x)(\exists y)[R(x, y)]$, we read it from left to right, but to understand it, it can help to start with an inner quantifier (again, similar to SQL, where we build the query from the inside out).

Exercise 3.3.7. Define $a|b$ using an existential quantifier. (You can use arithmetic relations: $+$, $-$, $*$, $/$.)

Exercise 3.3.8. You are allowed to use the predicates $\text{prime}(x)$ and $a|b$ to express the following properties. (You can use arithmetic relations.)

1. (S) “ x is even”,
2. “ x is odd”,
3. “ x is the sum of two odd numbers”,
4. “Every even number greater than 4 is the sum of two odd primes”. (This is known as Goldbach’s conjectures; its truth has not been established yet.)

Exercise 3.3.9. For each of the following four formulas find a binary relation R that makes the formula true and a relation R that makes it false.

1. $(\exists x)(\forall y)[R(x, y)]$,
2. $(\forall x)(\exists y)[R(x, y)]$,
3. $(\exists y)(\forall x)[R(x, y)]$,
4. $(\forall y)(\exists x)[R(x, y)]$.

Exercise 3.3.10. State in plain English what the following formulas express:

1. (S) $(\exists x)(\forall y)[x \leq y]$,
2. $(\forall x)(\exists y)[x \leq y]$,
3. $(\exists y)(\forall x)[x \leq y]$,

$$4. (\exists x)(\forall y)[x \leq y].$$

Also, for each of the formulas from the previous exercise decide whether it is true or not if interpreted in the universe of (a) the natural numbers, \mathbb{N} , (b) the integers, \mathbb{Z} , (c) the real numbers \mathbb{R} .

Example 3.3.11. We saw how to define primality using a universal quantifier. A number n is prime if and only if

$$n > 1 \wedge (\forall k)[k|n \rightarrow (k = 1 \vee k = n)].$$

You will also often find a prime number defined as a number larger than 1 that has no divisors other than 1 and itself. Using an existential quantifier, this could directly be expressed as

$$n > 1 \wedge \overline{(\exists k)[k|n \wedge k \neq 1 \wedge k \neq n]}.$$

The two definitions are identical; think about it: saying that every divisor of n is either 1 or n is the same as saying that there is no divisor of n which is different from 1 and n .

The previous example was a special case of the following powerful observation:

$$(\forall x)[P(x)] \leftrightarrow \overline{(\exists x)[\overline{P(x)}]},$$

which says that a property P is always true if and only if there is no counterexample, that is no x that makes it false. As usual, there is a dual way of expressing the relationship between universal and existential quantification:

$$(\exists x)[P(x)] \leftrightarrow \overline{(\forall x)[\overline{P(x)}]},$$

which says that there is an x that makes P true if and only if P is not false for all x .

These should look familiar: they are DeMorgan's laws in yet another device; if you think of \forall as a huge \wedge over all elements of the universe, and \exists as a \vee over all elements of the universe, the analogy should become clear.⁴ It means that we do not need both \forall and \exists in logical formulas, we can always eliminate one. For practical reasons, we use both; but recall SQL, for example; there is an EXISTS quantifier (which does not quite correspond to \exists but comes close), but there is no FORALL or ALL. Which is, why expressing conditions that involved universal quantifiers are so painful to write in SQL. Recall the query:

```
SELECT Department, CourseNr, CourseName
FROM Course
WHERE NOT EXISTS (
    SELECT StudentID
```

⁴Indeed, the notation \bigwedge_x for $\forall x$ and \bigvee_x for $\exists x$ are common in more algebraically oriented logic texts.

```

FROM Enrolled
WHERE CourseID = CID
EXCEPT
SELECT SID
FROM Student
WHERE Program = 'COMP-SCI');

```

It listed all courses that are only taken by computer science students; that is, all students enrolled in the course are computer scientists. Easy to say in English; in SQL we had to say: there are no students which are in the course, and which are not computer scientists.

Example 3.3.12. Let us work an extended example from a slightly different domain; we want to reconstruct the famous epsilon/delta definition of continuity in calculus. What does it mean for a function to be continuous? It means it does not make any jumps anywhere in its graph; compare, for example, the Heaviside function $H(x) = 0$ for $x < 0$ and $H(x) = 1$ for $x \geq 0$ to the parabola $f(x) = x^2$.

H is not continuous because it jumps at 0. This is our first break into the problem: we will call a function continuous, if it is continuous at all its points. So from now on, let us concentrate on a particular x value, call it x_0 . For a function f to be continuous at x_0 it cannot jump at x_0 . How can we capture what it means for f to jump at x_0 ? It means that f , on at least one side of x_0 does not come close to $f(x_0)$, that is, there is a $\varepsilon > 0$ such that $f(x)$ and $f(x_0)$ differ by at least ε on that side, even, if x gets arbitrarily close to x_0 . In other words, there is no small neighborhood of x_0 such that the values of $f(x)$ in that small neighborhood of x_0 are within a distance of at most ε from $f(x_0)$. Let us make “neighborhood” precise: we mean points within a distance of at most δ from x_0 . So being discontinuous at x_0 means that for some $\varepsilon > 0$ there is no $\delta > 0$ such that

$$|x - x_0| < \delta \rightarrow |f(x) - f(x_0)| < \varepsilon.$$

Or, using quantifiers:

$$(\exists \varepsilon > 0) \overline{(\exists \delta > 0)[|x - x_0| < \delta \rightarrow |f(x) - f(x_0)| < \varepsilon]}.$$

In other words, being continuous at x_0 can be defined as

$$(\forall \varepsilon > 0)(\exists \delta > 0)[|x - x_0| < \delta \rightarrow |f(x) - f(x_0)| < \varepsilon].$$

To express that a function is continuous at all points, we would have to write

$$(\forall x_0)(\forall \varepsilon > 0)(\exists \delta > 0)[|x - x_0| < \delta \rightarrow |f(x) - f(x_0)| < \varepsilon].$$

We have seen how to express existence and universal truth. These are powerful tools to express many other properties. For example, we already saw how to express uniqueness; we can write there is at most one x such that $P(x)$ as

$$(\forall x)(\forall y)[P(x) \wedge P(y) \rightarrow x = y,$$

as we saw in the social security example. Now that we understand the relation between existential and universal quantification, we see that this is the same as saying

$$\overline{(\exists x)(\exists y)[P(x) \wedge P(y) \wedge x \neq y]}.$$

You will sometimes find $(\exists^{\leq 1})[P(x)]$ for this quantifier. In this notation, $\exists^{\geq 1}$ is just the same as \exists . More common is the version $\exists!$ which means “there is exactly one”, which could also be written $\exists=1$.

Exercise 3.3.13. Express that every American citizen has a unique social security number (so you have to express that they do have such a number and that the number is unique) using the relation $R(x, y) =$ “ x has social security number y ” and $Q(x) =$ “ x is American”.

Example 3.3.14. Uniqueness sometimes occurs in database queries; for example, we might be asked to list all students that are members in at most one student group. Remember Maass’ advice: consider the negated version, which would be students that are members in at least two groups:

```
SELECT *
FROM Student
WHERE EXISTS (
  SELECT *
  FROM MemberOf AS M1, MemberOf AS M2
  WHERE M1.StudentID = SID AND M2.StudentID = SID AND
        NOT (M1.GroupName = M2.GroupName);
```

which lists all students for whom there are two different records of membership in student groups. What we really want is the opposite: students for which this is not the case:

```
SELECT *
FROM Student
WHERE NOT EXISTS (
  SELECT *
  FROM MemberOf AS M1, MemberOf AS M2
  WHERE M1.StudentID = SID AND M2.StudentID = SID AND
        NOT (M1.GroupName = M2.GroupName);
```

As we saw in the case of quantifiers, unique existence is expressed by saying there are not two different examples (which includes the case that there are none).

Of course, using counting, there is an easier way to achieve the same result:

```
SELECT SID, LastName, FirstName
FROM Student, MemberOf
WHERE SID = StudentID
GROUP BY SID, LastName, FirstName
HAVING count(*) <= 1;
```


This version is more convenient, of course, since it easily generalizes to “at most 2”, “at most k ”, “exactly k ”, and so on. Nevertheless, you will encounter the first version.

Exercise 3.3.15. Reexpress $(\exists=2)[P(x)]$, meaning, there are exactly two x which make P true using only regular \exists and \forall quantifiers.

One final example: how would we formally state Euclid’s theorem, which says that there are infinitely many primes?

$$(\exists^\infty)[\text{prime}(x)].$$

Using \exists^∞ amounts to cheating, of course; how can we express this using the tools we have? We could try to say: there is a set, it only contains prime, and the set is infinite. This would lead us to second-order logic and we would still have to express infinity. Alternately, we might be tempted to write something like:

$$(\exists x_1)(\exists x_2)\dots[x_1 \neq x_2 \wedge \dots \wedge \text{prime}(x_1) \wedge \text{prime}(x_2)\dots],$$

that is we would use an infinite number of quantifiers and a formula of infinite lengths. Indeed, logic allowing such formulas exist. For the prime numbers, however, there is an easier solution, just using a small number of our regular quantifiers.

Exercise 3.3.16. Express that there are an infinite number of primes using only \exists and \forall and $\text{prime}(x)$.

3.4 More on Relations

We have seen ordering and equivalence relations, but there is a third class of relations that is fundamental to mathematics that we have not mentioned yet: functions. The concept of function underwent some significant changes, mostly through development in the theory of the integral calculus; at this point a function, for a mathematician, is pretty much anything that associates a single output value with each input value. In particular, we observe that a function is a relation, indeed a binary one. There are two properties a binary relation has to fulfill to be a function: it must have at least one output value for each input value, and it must have at most one output value for each input value.

Formally, a binary relation R is called *single-valued* if

$$(\forall x)(\forall y)(\forall z)[R(x, y) \wedge R(x, z) \rightarrow y = z].$$

(This should look familiar.) It is called *total* if

$$(\forall x)(\exists y)[R(x, y)].$$

A single-valued and total binary relation R is called a *function*; since by definition for every x there is a unique y such that $R(x, y)$ we can write $f(x) = y$ in the function notation.

Example 3.4.1. The relation $R(x, y) = "x + y = 2"$ is single-valued and total, so it defines a function, namely $f(x) = y = 2 - x$. On the other hand, the relation $R(x, y) = "x = y^2"$ is neither total nor single-valued over the reals, so it does not define a function there. Over the complex numbers, the relation becomes total, but it is still not single-valued ($3^2 = (-3)^2 = 9$).

If $f \subseteq X \times Y$ is a function, we write it as $f : X \rightarrow Y$, and call X the *domain* of f and Y the *range*.

We also write $f(X) = \{f(x) : x \in X\}$, the *image* of f . If $f(X) = Y$, we call f *onto* or *surjective*, that is every element of the domain is taken on as a function value. For example, the function $f : \mathbb{N} \rightarrow \mathbb{N}$ that takes a number and removes its first digit is onto; e.g. $f(1923) = 923$. (What about the function that removes the last digit?).

For example, the function $f : \mathbb{N} \rightarrow \mathbb{N}$ that maps a number to its lengths in digits is nearly surjective: there is no number with no digits, so 0 is not in $f(\mathbb{N})$. So the function $f : \mathbb{N} \rightarrow \mathbb{N} - \{0\}$ that maps a number to its lengths in digits is surjective.

If $f(x) = f(y)$ implies that $x = y$ for all $x, y \in X$, then f is *one-to-one* or *injective*. Consider, for example, the function taking Americans with a social security number to their social security number. One-to-oneness expresses uniqueness.

Exercise 3.4.2. Which of the following functions are one-to-one and/or onto?

1. $f : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto x$
2. $f : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto x^2$
3. $f : \mathbb{R} \rightarrow \mathbb{R}^{\geq 0}, x \mapsto x^2$, where $\mathbb{R}^{\geq 0}$ are the positive real numbers.
4. $f : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto x^3$

Since any function f is a relation, we can write f^{-1} for the inverse relation to f that is, in the set-theoretic view:

$$\{(y, x) : y = f(x), x \in X\}.$$

If f is onto, then for every y there is an x such that $(y, x) \in f^{-1}$. If, moreover, f is one-to-one, then there is exactly one y such that $(y, x) \in f^{-1}$. In other words, f^{-1} is total and single-valued, so it is a function itself. This observation deserves a definition: a function f which is one-to-one and onto is called *bijective*. Our observation showed that a bijective function can always be inverted:

Lemma 3.4.3. *If $f : X \rightarrow Y$ is bijective, then f^{-1} is a function from Y to X . f^{-1} is also bijective.*

Exercise 3.4.4. Prove the lemma. In particular show that f^{-1} is total, single-valued, one-to-one and onto. How do these properties related to the same properties for f ?

A bijection $f : X \rightarrow Y$ shows that X and Y are really the same set—up to renaming the elements of the set.

Example 3.4.5. Here is a simple bijection from $\{\text{Homer, Marge, Lisa, Bart, Maggie}\}$ to $\{1, 2, 3, 4, 5\}$; let $f(\text{Bart}) = 1$, $f(\text{Lisa}) = 2$, $f(\text{Marge}) = 3$, $f(\text{Homer}) = 4$, $f(\text{Maggie}) = 5$.

The example shows one of the shortcomings of bijections: they do show that two sets are really the same up to renaming of elements, but they do not maintain any other sort of structure inherent in the set. For example, the set $\{\text{Homer, Marge, Lisa, Bart, Maggie}\}$ did not have any natural order. However, the set $\{1, 2, 3, 4, 5\}$ does. So we can use a bijection with a subset of the natural numbers to order an unordered set. Looked at in the opposite direction, the set $\{1, 2, 3, 4, 5\}$ lost its order when renamed $\{\text{Homer, Marge, Lisa, Bart, Maggie}\}$. In mathematics and computer science you therefore often need a stronger type of bijections, called isomorphisms that maintain the structure between the sets.

- Exercise 3.4.6.**
1. How many different bijections from $\{\text{Homer, Marge, Lisa, Bart, Maggie}\}$ to $\{1, 2, 3, 4, 5\}$ are there?
 2. If there is a bijection between $\{\text{Homer, Marge, Lisa, Bart, Maggie}\}$ and a set X , what can we say about the set X ?

The functions we have discussed so far are *unary* (as functions, not as relations, as relations they are binary), in that a single input determines a single output. What about functions like addition: $5 + 7 = 12$. We can think of this as $+(5, 7) = 12$ (and there are programming languages that do), that is a binary function $+$ which takes two summands and compute their sum as the output. As a relation, this binary function $+$ is a subset of \mathbb{R}^3 , namely $f = \{(x, y, z) : x + y = z\}$. That is $+$ has \mathbb{R}^2 as its domain, and \mathbb{R} as its range. In this way we can easily understand functions with multiple arguments.

Excursion: Databases and Functions

Databases naturally support functions, falling into two categories: aggregate functions which accumulate information over multiple records, such as `count` and functions used on field-values such as addition and multiplication.

For example,
`SELECT avg(started)`
`FROM Student;`

gives us the average year that students started. Or consider

```
SELECT min(year), max(year)
FROM Course, Enrolled
WHERE CID = CourseID AND
      CourseName = 'Theory of Computation';
```

which lists the first and last year that a course named 'Theory of Computation' has been offered. You can also sum values using the `sum` function.

There are built-in functions for arithmetic, `+`, `*`, `-`, `/`, together with other mathematical functions, e.g. `abs`, `sin`, `mod`, `ceiling`, `floor` that might come in handy. There are also special functions for strings (words). E.g. `concat` allows you to concatenate strings. So if you want to output the names of your students looking like "Brennigan, Marcus" you could do so by saying:

```
SELECT concat(lastname, ', ', firstname)
FROM student;
```

(Other systems will allow `+` or `&` for concatenation.)

Example 3.4.7. Let us say we want a formatted list of students and what year they are in. We know when each student started, so all we have to do is to subtract that year from the current year and add 1. E.g. if the student started in 2005 and it is now 2007, then this is their $2007 - 2005 + 1 = 3$ rd year. In H2 the current date is in a variable named `current_date`. We do not actually want the full current date, but just the year; so we use the built-in function `year` to extract the year information from `current_date`: `year(current_date)`. Then all we have to do is add the arithmetic:

```
SELECT concat(lastname, ', ', firstname),
        concat('Year: ', year(current_date) - started + 1)
FROM student;
```

The different database systems vary wildly on the names of built-in functions and the details of how they are used, so if you are using this query in a system other than H2 it will need some rewriting; the basic structure will be the same though.

- Exercise 3.4.8.**
1. List the students that have been at the university longest.
 2. List the junior student group president(s), that is the presidents that got appointed most recently.
 3. For each student list how many years it has been since they have taken their last course.
 4. For each student list how many quarters they have been at the university (each year has three quarters, Fall quarter lasts from September to November, Winter from January to March, and Spring from April to June). For example, a student that started in 2006 (we assume all students start in fall) has been at the university for four quarters in September 2007.

The following result regularly makes it into the top ten lists of the most beautiful results in mathematics.

Theorem 3.4.9 (Cantor). *There is no onto function from a set to its powerset.*

Proof. Let the set in question be X and suppose there is an onto function $f : X \rightarrow \mathbf{P}(X)$. Define a set A as follows:

$$A = \{x \in X : x \notin f(x)\}.$$

Then A is a subset of X , i.e. $A \subseteq X$. Furthermore, the way we defined it, $A \neq f(x)$ for all x . Why? Well, suppose $A = f(x)$ for some x . But if $x \in f(x)$ if and only if $x \notin A$, so the two sets differ on x . In other words, f is not onto. ■

The proof is known as *Cantor's diagonal argument*: you can visualize the proof as an infinite matrix. The rows of the matrix are labeled with elements of X , the columns with elements of $\mathbf{P}(X)$ listed as $f(x)$ for $x \in X$. The proof constructs a new set A by letting x be in A if and only if $x \notin f(x)$, so the set is constructed using the diagonal of the matrix.

There are strong similarities to Russell's paradox.

Corollary 3.4.10. *There is no one-to-one function from $\mathbf{P}(X)$ to X .*

Proof. Suppose there was a one-to-one function g from $\mathbf{P}(X)$ to X , that is if $g(A) = g(B)$, then $A = B$ for all $A, B \subseteq X$. Define a new function $f : X \rightarrow \mathbf{P}(X)$ as follows: on input x , if there is a set A such that $g(A) = x$, then let $f(x) = A$. Otherwise, let $f(x) = \emptyset$. Note that f is indeed a function from X to $\mathbf{P}(X)$. We claim that this function is onto. The reason is that for any set $A \subseteq X$, there is an x such that $f(x) = A$, namely $g(A)$: $f(g(A)) = x$ by definition. ■

A very simple and short proof. What earned it its reputation? Maybe two things: the diagonalisation method (while having been around before Cantor) found its purest form and a powerful application. To understand the application we first need to step back a little.

Suppose there is a bijection f between two sets A and B . In a sense, A and B are the same set then, just giving different names to objects: $x \in A$ is called $f(x) \in B$. The existence of a bijection between two sets establishes an equivalence relation between sets.

Exercise 3.4.11. Verify that the relation $R(A, B)$ which is true of there is a bijection from A to B is an equivalence relation.

What are the equivalence classes of this relation? Think small: what kind of set can you put into a bijection with one of your fingers? With five of your fingers? It looks as if the equivalence classes of the relation R are *cardinalities* of sets, that is, the sizes of a set. Indeed, this is how Cantor defined the notion of cardinality. We use $|A|$ for the cardinality of a set. Then we define

$$|A| = |B|$$

if and only if there is a bijection from A to B . How do we define $|A| \leq |B|$? As you would expect: $|A| \leq |B|$ if and only if there is a one-to-one function from A to B . That means A can be embedded in B without losing any information about A because any two elements of A map to different elements of B . In a sense A , as $f(A)$ has become part of B . That means that B is at least as large as A .

Theorem 3.4.12 (Cantor, Schroeder, Bernstein). *If $|A| \leq |B|$ and $|B| \leq |A|$ then $|A| = |B|$.*

As you can see from the list of names this is not a trivial theorem (do not confuse $=$ and \leq with the symbols we use to compare natural numbers or real numbers, we defined the meaning of these symbols anew, so we have to prove something that would be obvious in a different context).

Now what Cantor's theorem shows is that $|A| < |\mathbf{P}(A)|$ because it states that $|\mathbf{P}(A)| \not\leq |A|$ because there is no one-to-one function from $\mathbf{P}(A)$ to A .

This leads to another paradox set theory is responsible for: is there a largest set? It seems the answer would have to be yes, namely the set containing everything. This set is sometimes known as V . What about $\mathbf{P}(V)$. If V really contains everything, it must contain all the elements of $\mathbf{P}(V)$, that is $\mathbf{P}(V) \subseteq V$. But then $|\mathbf{P}(V)| \leq |V|$, contradicting Cantor's theorem.

We conclude this section with a thought experiment known as Hilbert's Hotel. Hilbert's hotel has rooms numbered 1, 2, 3, and so on without end. (You might ask where there might be room for such a hotel; making appropriate assumptions about space, you can fit it pretty much anywhere you want.) A new guest has arrived, but all the rooms are taken. Where can we put the new guest? The solution is easy: every guest moves down one room (from n to $n+1$), freeing up the first room.

Exercise 3.4.13. What would you do with a group of k guests?

Even if there was an infinite number of guests, g_1, g_2, g_3 , and so on, we could make room for them: move the current guest in room n to room $2n$, and move $g_1, g_2, g_3 \dots$ into rooms 1, 3, 5, \dots

If we rephrase these observations in terms of cardinality, we see that

- $|\mathbb{N}| = |\mathbb{N} - \{1\}|$,
- $|\mathbb{N}| = |\mathbb{N} - \{1, 2, \dots, k\}|$,
- $|\mathbb{N}| = |2\mathbb{N}| = |2\mathbb{N} + 1|$, where $2\mathbb{N} := \{2n : n \in \mathbb{N}\}$ is the set of even numbers and $2\mathbb{N} + 1 := \{2n + 1 : n \in \mathbb{N}\}$ is the set of odd numbers.

These results might be counterintuitive, in particular the last one. Cardinalities of infinite sets behave differently from finite sets, in particular the old dictum that the part is smaller than the whole is no longer true. But maybe that should not surprise us.

3.5 Exercises

1. Explain what went wrong in the following conversation in a bookstore:

“Do you have any books by Bulgakov?”

“Yes, we do, check in fiction under B”

“I can’t seem to find his Molière novel.”

“Oh, we don’t carry that one.”

“But didn’t you say you had *any* book by Bulgakov?”

2. In §281 of his *Parerga and Paralipomena*, Schopenhauer writes

Schon Rousseau hat in der Vorrede zur “Neuen Heloise” gesagt: “Jeder ehrliche Mann setzt seinen Namen unter das, was er schreibt”, und allgemein bejahende Stze lassen sich per contrapositionem umkehren.

In, free, translation:

Rousseau already noted in the preface to his *The New Heloise*: “Ever honest person puts his name under what he writes.”, and universal sentences can be reversed by contraposition.

- (i) Express Rousseau’s statement formally, using $H(x) = “x \text{ is honest}”$ and $S(x) = “x \text{ puts his name under what he writes}”$.
 - (ii) Explain what Schopenhauer was implying by his jibe about reversing universal sentences by contraposition.
 - (iii) Capture Rousseau’s statement even more precisely, using $R(x, y) = “x \text{ writes } y”$. $H(x) = “x \text{ is honest}”$ and $S(x, y) = “x \text{ puts his name under } y”$.
3. Talking about Nietzsche, in 1913, Mencken writes:
- No reader of current literature [...] can have failed to notice the increasing pressure of his ideas.
- (i) Express Mencken’s statement formally, using $R(x) = “x \text{ is a reader of current literature}”$ and $F(x) = “x \text{ has noticed the increasing pressure of his ideas}”$.
 - (ii) If you transcribed Mencken’s sentence accurately in (i), you used a double negation. Can you express what he was saying positively, without using any negation?
4. In calculus we write $\lim_{x \rightarrow \infty} f(x) = c$ to say that $f(x)$ tends towards c as x goes to infinity, or, in other words, $|f(x) - c|$ becomes arbitrarily small as x grows larger. More precisely, we require that we can find an x' for every $\varepsilon > 0$ such that $|f(x) - c| < \varepsilon$ for all $x \geq x'$. Which of the following formal versions expresses this correctly? Argue that your answer is correct.

- (a) $(\exists x')(\forall \varepsilon > 0)(\forall x \geq x')[|f(x) - c| < \varepsilon]$,
- (b) $(\forall \varepsilon > 0)(\exists x')(\forall x \geq x')[|f(x) - c| < \varepsilon]$,
- (c) $(\forall \varepsilon > 0)(\forall x')(\forall x \geq x')[|f(x) - c| < \varepsilon]$,
- (d) $(\exists \varepsilon > 0)(\exists x')(\forall x \geq x')[|f(x) - c| < \varepsilon]$,
- (e) $(\exists x')(\exists \varepsilon > 0)(\forall x \geq x')[|f(x) - c| < \varepsilon]$,

Note how this allows us to eliminate infinity from calculus.

5. Fermat's last theorem says that $x^n + y^n = z^n$ has no solutions $x, y, z \in \mathbb{N}$ for any $n > 2$ other than the trivial ones (in which one of x, y or z is zero). State this result formally using quantifiers.
6. We saw how to express a condition of the type $A \subseteq B$ in a database by using NOT EXISTS B EXCEPT A. In this exercise we will see another way of achieving the same result that does not use double negation.
 - (i) Argue that $|A \cap B| = |A|$ if and only if $A \subseteq B$.

Use the observation from (i) to write queries for the following problems.

- (ii) List courses which only computer science students ever enrolled in.
- (iii) List courses that only graduate students have enrolled in.
- (iv) List all courses that have been taught every year (that courses have been taught).

Chapter 14

Hints

14.1 Chapter 1

Exercise 1.1.3 All tables have 8 rows, the second table has two columns, the other two tables have one column.

Exercise 1.1.4 Here are the outputs you should see:

SSN
null
123123123
111111111
321321321

FirstName
Deepa
Prakash

LASTNAME	FIRSTNAME	SID
Patel	Deepa	14662
Johnson	Peter	32105
Patel	Prakash	75234
Snowdon	Jennifer	93321

LASTNAME	FIRSTNAME	SID
Brennigan	Marcus	90421
Patel	Deepa	14662
Starck	Jason	19992
Winter	Abigail	11035
Patel	Prakash	75234

Exercise 1.2.1 The number of records returned by the queries are: 1. 2 records, 2. 2 records, 3. 2 records, 4. 2 records, 5. 1 record, 6. 3 records.

Exercise 1.2.3 The number of records returned by the queries are: 1. 6 records, 2. 2 records, 3. 1 record, 4. 4 records, 5. 6 records, 6. 1 record, 7. 1 record, 8. 0 records.

Exercise 1.3.2 1. They differ. 2. They differ.

Exercise 1.4.5 Use `SELECT DISTINCT` for all of these queries, since you want a list of students, duplicates would be meaningless. With duplicates removed, the number of records in the output to the queries is as follows: 1. 3 records, 2. 2 records, 3. 2 records, 4. not possible yet, 5. 1 record.

Exercise 1.5.1 1. Even number means 0 or 2 in this case, that is, either all three propositional variables have to be false, or exactly one of them is false.

Exercise 1.5.4 If you follow the procedure, your solution will be a conjunction of three clauses (which are disjunctions of literals). If you use transformations or simplify your solution, you can obtain a formula in DNF which is the conjunction of two clauses, each of which contains two variables.

Exercise 1.5.9 1. Consider $p \uparrow p$.

Exercise 1.5.11 Convert the formulas to DNF, and then use the results from the previous exercise.

Exercise 2.2.4 The main table for the second query is `StudentGroup`, not `Student`. Concentrate on getting the list of IDs. (This will be a nested query.) With the outer query for presentation, the solution will be a doubly nested SQL query.

Exercise 2.6.1 For the first question find the student groups that *do* have members first. For the fifth question (tricky), do *not* use `EXCEPT`, but rephrase the query using propositional logic instead of set operations.

Exercise 2.6.3 For the first question, concentrate on a particular student: what courses is the student enrolled in (set A), and what are the CSC courses (set B)? The requirement translates to $A \subseteq B$.

Exercise 3.1.3 Here is the definition of sibling from Merriam-Webster: “one of two or more individuals having one common parent”.

Exercise 3.1.14 These properties are directly due to the corresponding properties of set membership.

Exercise 3.1.22 If x is brother to y , what is y to x ? Moreover, don't forget that we know something about x .

Exercise 3.3.7 Division does not help at all. Multiplication does.

14.2 Chapter 10

Exercise 10.0.13 Computing the last digit is the same as calculating the number modulo 10.

Chapter 15

Solutions To Selected Exercises

15.1 Chapter 1

Exercise 1.1.4 The query for the first question is

```
SELECT SSN
FROM Student
WHERE Career = 'GRD';
```

The query for the third question is

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Program = 'COMP-SCI';
```

Exercise 1.2.1 The query for the first question is

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Career = 'GRD' AND Program = 'COMP-SCI';
```

The query for the fourth question is

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Program = 'COMP-SCI' AND NOT City = 'Chicago';
```

Exercise 1.2.3 The query for the second question is

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Started = 2001 OR Started = 2002;
```

The query for the sixth question is

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Career = 'GRD' AND SSN is null;
```

Exercise 1.3.2 1. They differ on $p = \top$ and $q = \top$.

Exercise 1.3.4 The equivalence of double-negation with assertion is proved by the following truth-table:

φ	$\bar{\varphi}$	$\bar{\bar{\varphi}}$	$\varphi \leftrightarrow \bar{\bar{\varphi}}$
\perp	\top	\perp	\top
\top	\perp	\top	\top

The idempotence of \wedge is shown by the following truth-table:

φ	$\varphi \wedge \varphi$	$\varphi \leftrightarrow \varphi \wedge \varphi$
\perp	\perp	\top
\top	\top	\top

Exercise 1.4.1 You'd need to add the following row to the Enrolled table.

StudentID	CourseID	Quarter	Year
32105	9219	Spring	2003

Exercise 1.4.5 Solution to first query.

```
SELECT DISTINCT LastName, FirstName, SID
FROM Student, Studentgroup
WHERE SID = PresidentID;
```

Exercise 1.4.6 First check constraint: the following line has to be added to the Course table.

```
CHECK (NOT CourseNr = '000')
```

Exercise 1.5.1 1. $(\bar{p} \wedge \bar{q} \wedge \bar{r}) \vee (p \wedge q \wedge \bar{r}) \vee (p \wedge \bar{q} \wedge r) \vee (\bar{p} \wedge q \wedge r)$.

Exercise 1.5.4 The simplified solution is $(p \wedge q) \vee (p \wedge \bar{r})$.

15.2 Chapter 2

Exercise 2.2.1 The following query lists presidents of student groups founded before 2000.

```
SELECT LastName, FirstName, SID
FROM Student
WHERE SID IN
  (SELECT PresidentID
   FROM StudentGroup
   WHERE Founded < 2000);
```

Exercise 2.2.2 The first query lists all students who are not members of HerCTI, the second all students who are members of some group other than HerCTI. There are two differences: the first query will list students who are not members of any student group (which are not listed by the second query), and the second query will list students that are members of HerCTI, as long as they are also member of some other group.

Exercise 2.2.4 Solution to the first query:

```
SELECT Name
FROM StudentGroup
WHERE EXISTS
  (SELECT StudentID
   FROM MemberOf
   WHERE Name = GroupName);
```

Exercise 2.2.6 Solution to the second query:

```
SELECT LastName, FirstName, SID
FROM Student
WHERE SID IN
  (SELECT PresidentID
   FROM studentgroup
   WHERE presidentID NOT IN
     (SELECT StudentID
      FROM MEMBEROF
      WHERE groupname = name));
```

Exercise 2.3.2 The solution to the first question is {Lisa}.

Exercise 2.3.6 (1): $A \cap A = A$. (6): $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

Exercise 2.5.3 (2): $A \cap \bar{A} = \emptyset$.

Exercise 2.6.1 The solution to the first query can be written as


```
(SELECT Name
 FROM StudentGroup)
EXCEPT
(SELECT GroupName
 FROM MemberOf);
```

Exercise 2.6.3 The solution to the first query is as follows:

```
SELECT LastName, FirstName, SID
FROM Student
WHERE NOT EXISTS (
  SELECT CID
  FROM Enrolled
  WHERE StudentID = SID
EXCEPT
  SELECT CID
  FROM Course
  WHERE Department = 'CSC');
```

Exercise 2.8.3 (1): $2^5 = 32$.

15.3 Chapter 3

Exercise 3.1.3 (1) (see definition in Hints): nobody is their own sibling, so reflexivity fails very strongly (the relation is anti-reflexive: $R(x, x)$ fails for all x); the relation is symmetric, since if x and y have at least one common parent, so do y and x . On the other hand, R is not transitive, since x and y could have a common parent, and y and z could have a common parent, without x and z having a common parent.

Exercise 3.1.10 The following query lists the first and last year for each program that a student started in it.

```
SELECT Program, min(started), max(started)
FROM Student
GROUP BY Program;
```

Exercise 3.1.13 Student groups that have no members:

```
SELECT Name, count(*)
FROM StudentGroup, MemberOf
WHERE GroupName = Name
GROUP BY Name
HAVING count(*) = 0;
```

Exercise 3.2.8 The transitive closure of “is child of” is “is predecessor of”.

Exercise 3.3.1 (1) expresses that for every x , if x has a social security number, then x is American. In plain English: everybody who has a social security number is American, that is, only Americans have social security numbers.

Exercise 3.3.5 (1) R is anti-reflexive:

$$(\forall x)[\overline{R}(x, x)].$$

(4) R is not reflexive:

$$\overline{(\forall x)[R(x, x)]}.$$

Exercise 3.3.8 Two possible ways to define even: x is even if and only if $(\exists y)[y = x + x]$, or simply $2|x$.

Exercise 3.3.10 (1) says that there is an x that is less than or equal to all y (the *same* x for all $y!$). In other words: there is a smallest element. This is true for the natural numbers, $x = 1$, but it is not true for either integers or real numbers.

15.4 Chapter 8

Exercise 8.1.9 Here's the queue `reached` during the run of breadth-first search together with the distance calculated for the elements popped.

<code>{s}</code> pop s , add a	<code>distance[s] = 0</code>
<code>{a}</code> pop a , add b and h not s	<code>distance[a] = 1</code>
<code>{b, h}</code> pop b , add c and d not a	<code>distance[b] = 2</code>
<code>{h, c, d}</code> pop h , add g and i not a	<code>distance[h] = 2</code>
<code>{c, d, g, i}</code> pop c , do not add b	<code>distance[c] = 3</code>
<code>{d, g, i}</code> pop d , add e and f not b	<code>distance[d] = 3</code>
<code>{g, i, e, f}</code> pop g , do not add h	<code>distance[g] = 3</code>
<code>{i, e, f}</code> pop i , add j and k not h	<code>distance[i] = 3</code>
<code>{e, f, j, k}</code> pop e , do not add d	<code>distance[e] = 4</code>
<code>{f, j, k}</code> pop f , do not add d	<code>distance[f] = 4</code>
<code>{j, k}</code> pop j , do not add i	<code>distance[j] = 4</code>
<code>{k}</code> pop k , add l and n not i	<code>distance[k] = 4</code>
<code>{l, n}</code> pop l , do not add k	<code>distance[l] = 5</code>
<code>{n}</code> pop n , add m and t not k	<code>distance[n] = 5</code>
<code>{m, t}</code> pop m , do not add n	<code>distance[m] = 6</code>
<code>{t}</code> pop t , found exit, return <code>true</code>	<code>distance[t] = 6</code>

15.5 Chapter 10

Exercise 10.0.8 For the first problem, $m = 3$ or $m = 6$ would work.

Exercise 10.0.9 $4 \not\equiv 5 \pmod{2}$, since 2 does not divide $4 - 5 = -1$, and similarly, $4 \not\equiv 5 \pmod{3}$, since 3 also does not divide $4 - 5 = -1$. Also, $4 \not\equiv 9088 \pmod{5}$, since $4 - 9088 = -9084$ is not a multiple of 5.

Exercise 10.0.12 $31 * 68 * 902 - 777 * 973 \equiv 1 * 2 * 2 - 0 * 1 \equiv 4 \equiv 1 \pmod{3}$

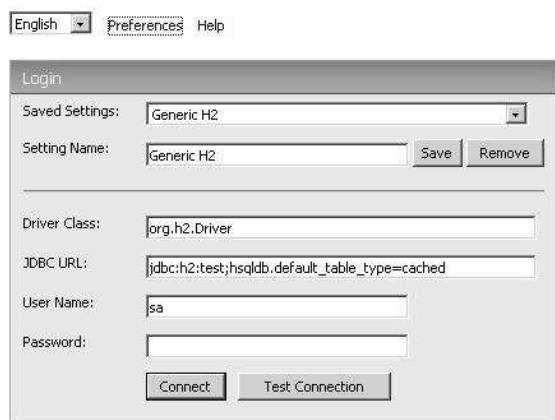
Exercise 10.0.14 A number x is divisible by 10 if and only if there is an integer c such that $x = 10c$. But if $x = 10c$, then $x \pmod{10} = 0$ and the last digit of x is 0. On the other hand if the last digit of x is 0, then $x \pmod{10} = 0$, that is $10|x$ and there is a c such that $x = 10c$.

Appendix A

Some Words on H2

If you want to run the queries using set intersection and difference, you will need a relational database system supporting `INTERSECT` and `EXCEPT`. A nice, easy-to-install package is the H2 database engine which you can download at <http://www.h2database.com>.

After installation run the H2 console (not in command line mode); this should bring up a browser window with the following login screen.



The screenshot shows a web browser window titled "Login" for the H2 database. At the top, there are links for "English", "Preferences", and "Help". The main content area includes a "Saved Settings" dropdown menu set to "Generic H2", a "Setting Name" text input field containing "Generic H2" with "Save" and "Remove" buttons, a "Driver Class" text input field with "org.h2.Driver", a "JDBC URL" text input field with "jdbc:h2:test;hsqldb.default_table_type=cached", a "User Name" text input field with "sa", and a "Password" text input field. At the bottom, there are "Connect" and "Test Connection" buttons.

Figure A.1: H2 login.

Click on the “connect” button, and you should see a window as in Figure A.2:

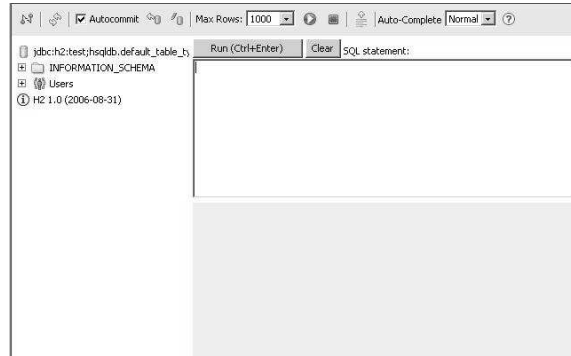


Figure A.2: H2 console.

You can use the window on the right to input SQL (below you will see the results. To create the university database, copy/paste the file “university.sql” into the window, and hit the “run” button. As a result you should see something like the screen in Figure A.3. You can now try running a simple query as in the

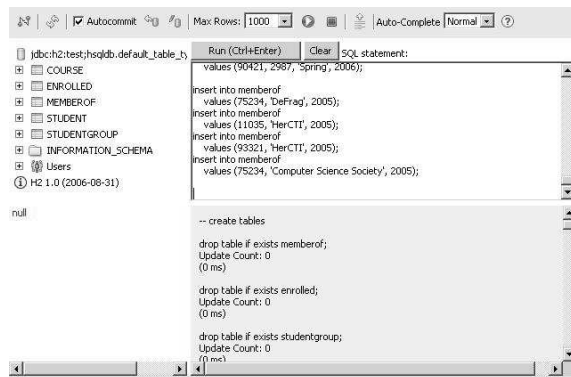


Figure A.3: H2 console with database.

screen shot in Figure A.4. To log out, hit the red symbol on the top left.

The screenshot shows the H2 console interface. The top toolbar includes options for 'AutoCommit', 'Max Rows: 1000', and 'Auto-Complete'. The left sidebar shows a tree view of the database schema, including tables like COURSE, ENROLLED, MEMBEROF, STUDENT, STUDENTGROUP, INFORMATION_SCHEMA, and Users. The main area contains a text input field with the SQL statement 'select * from student;' and a 'Run (Ctrl+Enter)' button. Below the input field, the results of the query are displayed in a table format.

LASTNAME	FIRSTNAME	SID	SSN	CAREER	PROGRAM	CITY	STARTED
Brennigan	Marcus	90421	987654321	UGRD	COMP-GPH	Evanston	2001
Patel	Deepa	14662	null	GRD	COMP-SCI	Evanston	2003
Snowdon	Jonathan	8871	123123123	GRD	INFO-SYS	Springfield	2005
Stark	Jason	19952	789789789	UGRD	INFO-SYS	Springfield	2003
Johnson	Peter	32105	123456789	UGRD	COMP-SCI	Chicago	2004
Winter	Abigail	11026	111111111	GRD	PHD	Chicago	2003
Patel	Prakash	75234	null	UGRD	COMP-SCI	Chicago	2001
Snowdon	Jennifer	93321	321321321	GRD	COMP-SCI	Springfield	2004

Below the table, it indicates '(8 rows, 0 ms)'.

Figure A.4: H2 console with results of query.

Appendix B

The University Database

Student

LastName	FirstName	SID	SSN	Career	Program	City	Started
Brennigan	Marcus	90421	987654321	UGRD	COMP-GPH	Evanston	2001
Patel	Deepa	14662	null	GRD	COMP-SCI	Evanston	2003
Snowdon	Jonathan	08871	123123123	GRD	INFO-SYS	Springfield	2005
Starck	Jason	19992	789789789	UGRD	INFO-SYS	Springfield	2003
Johnson	Peter	32105	123456789	UGRD	COMP-SCI	Chicago	2004
Winters	Abigail	11035	111111111	GRD	PHD	Chicago	2003
Patel	Prakash	75234	null	UGRD	COMP-SCI	Chicago	2001
Snowdon	Jennifer	93321	321321321	GRD	COMP-SCI	Springfield	2004

Enrolled

StudentID	CourseID	Quarter	Year
11035	1020	Fall	2005
11035	1092	Fall	2005
75234	3201	Winter	2006
08871	1092	Fall	2005
90421	8772	Spring	2006
90421	2987	Spring	2006

Course

CID	CourseName	Department	CourseNr
1020	Theory of Computation	CSC	489
1092	Cryptography	CSC	440
3201	Data Analysis	IT	223
9219	Desktop Databases	IT	240
3111	Theory of Computation	CSC	389
8772	Survey of Computer Graphics	GPH	425
2987	Topics in Digital Cinema	DC	270

MemberOf

StudentID	GroupName	Joined
75234	DeFrag	2005
11035	HerCTI	2004
93321	HerCTI	2005
75234	Computer Science Society	2002

StudentGroup

Name	PresidentID	Founded
Computer Science Society	75234	1999
Robotics Society	null	1998
HerCTI	93321	2003
DeFrag	90421	2004

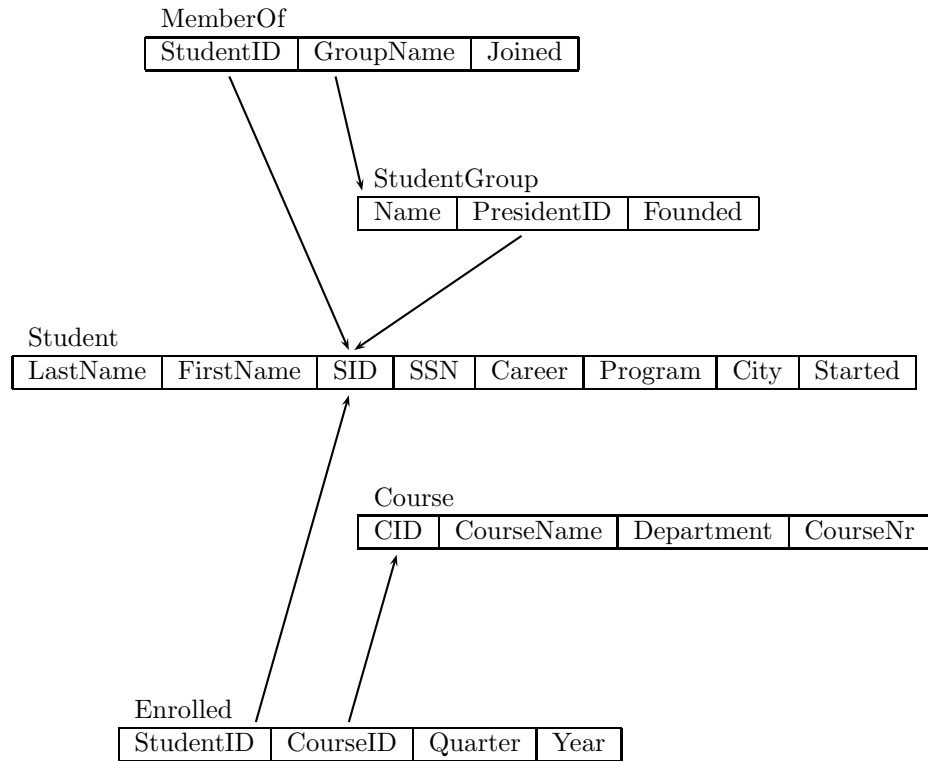


Figure B.1: Relationships in the University database

Appendix C

Fundamentals of SQL

The simplest SQL queries consist of a `SELECT` and a `FROM` clause.

```
SELECT LastName, FirstName
FROM Student;
```

In the `FROM` clause you specify which table your rows come from, in the `SELECT` clause you select the attributes of the rows that will get listed. Using `SELECT DISTINCT` in place of `SELECT` removes duplicate rows in the output.

```
SELECT DISTINCT LastName
FROM Student;
```

There is a third clause, the `WHERE` clause which allows you to specify requirements that a row needs to fulfill to get listed.

```
SELECT LastName, FirstName
FROM Student
WHERE Career = 'UGRD';
```

The conditions in the `WHERE` can be *atomic*, including comparisons such as `Career = 'UGRD'`, `year <= 1969`, `LastName < 'P'`, or they can be *compound* that is, Boolean combinations of atomic queries, such as `Career = 'UGRD' AND LastName < 'P'`. There are several special conditions built into SQL, including `is null` and `EXISTS` (which leads to a nested query).

```
SELECT LastName, FirstName
FROM Student
WHERE not SSN is null;
```

If you choose multiple tables in the `FROM` clause, SQL will generate all possible combinations of rows from each table. For example, the query

```
SELECT LastName, FirstName, Name
FROM Student, StudentGroup;
```

generates 8*4 rows of output, most of them not corresponding to meaningful information. Use the **WHERE** clause, connecting foreign key and primary key of the tables to restrict the output to meaningful rows only:

```
SELECT LastName, FirstName, Name
FROM Student, StudentGroup
WHERE PresidentID = SID;
```

Output of a SQL query can be grouped by any attribute; this means that rows for which that attribute has the same value get collapsed into a single row. For example,

```
SELECT City
FROM Student
GROUP BY City
```

lists all cities that students are from. In a grouped query you can only select attributes by which you have grouped, or aggregate functions of other attributes.

```
SELECT City, min(Started)
FROM Student
GROUP BY City
```

lists all cities, and, for each city, the earliest year that a student from that city begin their studies.

Conditions that need to be applied *after* grouping and aggregation need to be included in the **HAVING** clause rather than the **WHERE** clause. For example,

```
SELECT City
FROM Student
GROUP BY City
HAVING count(*) >= 3
```

lists all cities from which there are at least three students.

Finally, the output of a query can be ordered by attributes using the **ORDER BY** clause. By default the ordering is increasing in the appropriate ordering (numerical or lexicographic).

Let us review these features in a single query containing all clauses:

```
SELECT City, count(*)
FROM Student
WHERE started < 2005
GROUP BY City
HAVING count(*) >= 2
ORDER BY City;
```

The database management system retrieves all rows in the table `Student`, `FROM`, restricting the output to those rows for which `started` is less than 2005, `WHERE`. It then aggregates those output rows into groups by the attribute `City`, `GROUP BY`, and only retains those groups in which at least two rows have been aggregated, `HAVING`. It outputs the groups as names of cities and counts of rows, `SELECT`, ordered lexicographically by `City`, `ORDER BY`.

Appendix D

Propositional Logic

We use Roman letters p, q, r, s , etc. to denote variables representing propositions and Greek letters φ, ψ , etc. to denote formulas. A formula is either a proposition or one of the following: $\bar{\varphi}$ (not φ , the negation of φ), $\varphi \vee \psi$ (φ or ψ , the disjunction of φ and ψ), $\varphi \wedge \psi$ (φ and ψ , the conjunction of φ and ψ), $\varphi \rightarrow \psi$ (φ implies ψ), or $\varphi \leftrightarrow \psi$ (φ is equivalent to ψ), where φ and ψ are formulas, and we use parentheses as needed. The meaning of the logical operations is explained through truth-tables.

φ	$\bar{\varphi}$
\perp	\top
\top	\perp

φ	ψ	$\varphi \wedge \psi$
\perp	\perp	\perp
\perp	\top	\perp
\top	\perp	\perp
\top	\top	\top

φ	ψ	$\varphi \vee \psi$
\perp	\perp	\perp
\perp	\top	\top
\top	\perp	\top
\top	\top	\top

φ	ψ	$\varphi \rightarrow \psi$
\perp	\perp	\top
\perp	\top	\top
\top	\perp	\perp
\top	\top	\top

φ	ψ	$\varphi \leftrightarrow \psi$
\perp	\perp	\top
\perp	\top	\perp
\top	\perp	\perp
\top	\top	\top

The following table lists the rules of precedence for the most common logical operators we have seen.

strongest	\neg
	\wedge
	\vee
	\rightarrow
weakest	\leftrightarrow

We list some fundamental rules of logic, some of them are well-known enough to have names.

$\varphi \leftrightarrow \overline{\overline{\varphi}}$	(Double-Negation)
$\varphi \wedge \overline{\varphi} \leftrightarrow \perp$	(Law of Contradiction)
$\varphi \vee \overline{\varphi} \leftrightarrow \top$	(Law of Excluded Middle)
$\varphi \vee \perp \leftrightarrow \varphi$	
$\varphi \vee \top \leftrightarrow \top$	
$\varphi \wedge \perp \leftrightarrow \perp$	
$\varphi \wedge \top \leftrightarrow \varphi$	
$\varphi \wedge \varphi \leftrightarrow \varphi$	(Idempotence of \wedge)
$\varphi \vee \varphi \leftrightarrow \varphi$	(Idempotence of \vee)
$\varphi \wedge (\psi \wedge \theta) \leftrightarrow (\varphi \wedge \psi) \wedge \theta$	(Associativity of \wedge)
$\varphi \vee (\psi \vee \theta) \leftrightarrow (\varphi \vee \psi) \vee \theta$	(Associativity of \vee)
$\varphi \wedge \psi \leftrightarrow \psi \wedge \varphi$	(Commutativity of \wedge)
$\varphi \vee \psi \leftrightarrow \psi \vee \varphi$	(Commutativity of \vee)
$\overline{\varphi} \wedge \overline{\psi} \leftrightarrow \overline{\varphi \vee \psi}$	(DeMorgan)
$\overline{\varphi} \vee \overline{\psi} \leftrightarrow \overline{\varphi \wedge \psi}$	(DeMorgan)
$\varphi \wedge (\psi \vee \theta) \leftrightarrow (\varphi \wedge \psi) \vee (\varphi \wedge \theta)$	(Distributivity of \wedge over \vee)
$\varphi \vee (\psi \wedge \theta) \leftrightarrow (\varphi \vee \psi) \wedge (\varphi \vee \theta)$	(Distributivity of \vee over \wedge)

Here are some of the laws governing implication:

$\varphi \rightarrow \varphi$	
$\perp \rightarrow \varphi$	(ex falso quodlibet ¹)
$(\top \rightarrow \varphi) \rightarrow \varphi$	
$(\varphi \rightarrow \perp) \rightarrow \overline{\varphi}$	(Proof by contradiction)
$(\varphi \wedge (\varphi \rightarrow \psi)) \rightarrow \psi$	(Modus Ponens)
$((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \theta)) \rightarrow (\varphi \rightarrow \theta)$	(Modus Barbara)
$(\varphi \wedge \psi) \rightarrow \varphi$	(\wedge -weakening)
$\varphi \rightarrow (\varphi \vee \psi)$	(\vee strengthening)
$(\varphi \rightarrow \psi) \rightarrow (\overline{\psi} \rightarrow \overline{\varphi})$	(contrapositive)
$((\varphi \rightarrow \psi) \rightarrow \varphi) \rightarrow \varphi$	(Peirce's law)

Appendix E

Problem Solving

In his *Grundriß der Logik*, Maass gives some sound advice on problem solving in a section on practical logic. It is well worth pondering.

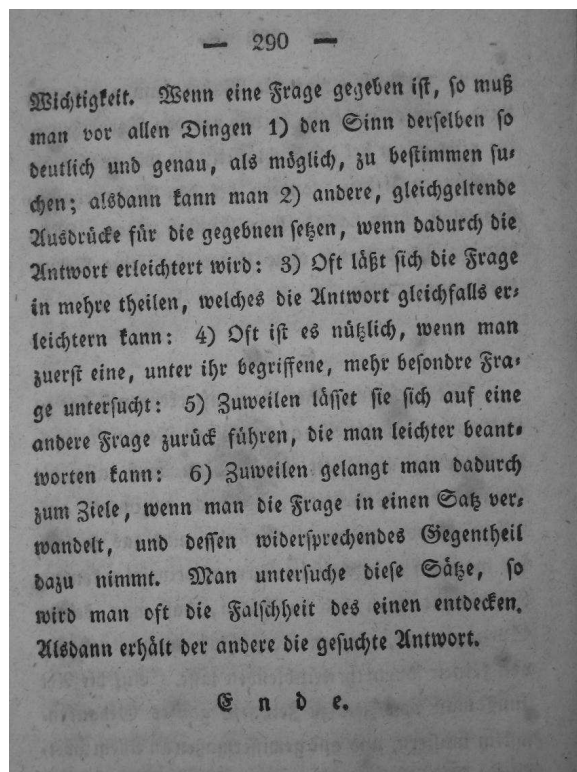


Figure E.1: Extract from Maass' *Grundriß der Logik*

The German text translates as

When given a question, one should first of all 1) determine its meaning as clearly and accurately as possible; then one may 2) replace terms with other terms of the same meaning if this simplifies finding the solution: 3) often, the question can be split into several questions, which can, similarly, simplify finding the answer: 4) it is often helpful to consider a special, more particular, version of the question: 5) sometimes the question can be reduced to another question which is easier to answer: 6) occasionally one obtains a solution by turning the question into a statement, and assuming its negation. On studying the negation one will often discover it is false, which obtains the answer to the original question.

Let us take each piece of advice one by one.

E.1 Determine the meaning of a question

This seems blatantly obvious, but often is anything but. Questions can be ambiguous, ill-phrased, even inconsistent. This might be due to the person asking the question, in which case you need to clarify the question. However, there are more fundamental problems that cloud the meaning of sentences. Natural languages are inherently ambiguous. Take, for example, the last sentence: did it say that all natural languages are inherently ambiguous, or just some of them; or most of them? In conversation the sentence could be made to have any of these meanings given the right context. Mathematics tries to avoid the pitfalls of natural languages by replacing it with formal languages, such as set theory and logic. However, this introduces new problems, as we will see: some technical terms, such as “and”, “or” and “if” sound like English words we know, but their assigned meaning is slightly different from the wide variety of meanings these words can take on in an English sentence. The other problem is that we cannot write everything using formal language only. While this would increase precision, it would decrease readability.

So, we take Maass’ first piece of advice to mean that a question should be read carefully, that all parts of it should be clearly understood, and that its meaning should be clear. The danger is not so much that you might not come up with an answer to the question, the danger is that your answer might be wrong, and you won’t be able to tell, or that your answer is right, but you don’t understand why. While luck is an ingredient of the problem solving process, you need to be able to evaluate and test your solutions.

E.2 Replace terms with equivalent terms

In a first step, replace a term by its definition. If you are asked to show that 11 is prime, you need to replace the word “prime” with its definition; then you

check that 11 fulfills that definition, that is, it has no divisors other than 1 and 11 and it is larger than 1.

Maass' advice is quite explicitly about terms and their definitions, but there is a broader way of viewing it: As you begin to understand a concept better and better you will accumulate operational knowledge about the concept; that is, you will know how to use the concept. For example, initially, when checking the primality of 11, you might have tried all possible divisors 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 to find that only 1 and 11 are divisors. After doing some examples, you quickly find out that you only have to check divisors up to $11/2$, that is 1, 2, 3, 4, 5. Indeed, it is enough to check up to $\sqrt{11}$, that is 1, 2, 3. If any number x at least $4 > \sqrt{11}$ was a divisor of 11, then $11/x < \sqrt{11}$ would also be a divisor in the range $1, \dots, \sqrt{x}$ that we check. As you see, we have already found two new ways to define primality, and there are many more. The application of the concept will determine which definition is the most useful; e.g. take the example of prime numbers again; if we were to write a program that checks whether a number is prime, we'd prefer the last definition, since it only requires us to check up to \sqrt{x} . However, in a mathematical proof, the original definition is much more useful.

E.3 Divide et impera

One of the core techniques in problem solving is to break the problem into smaller pieces (“divide”: divide), solve those pieces separately (sometimes by breaking them into even smaller pieces), and then combine the solutions into a solution of the original problem (“impera”: rule). The quote is often used in describing Caesar's strategy for conquering Gaul: piece by piece until he had conquered all of Gaul.¹

This maxim has application at many levels: imagine, for example, that you have to show the equivalence of two statements p and q . We know that $p \leftrightarrow q$ is the same as $p \rightarrow q$ and $q \rightarrow p$ so we can show each of the implications (the smaller problems) and conclude that p and q are equivalent. Or, to take another example from database queries: we separate the presentation aspects from the logical aspects. And at the logical level we distinguish between conditions that are immediate (through foreign key constraints) and those conditions that capture the logical core of the problem. Or do you remember Rubik's Cube? Solutions to Rubik's Cube typically work by solving the first, second and third layer of the cube separately.

E.4 Reduction

Reducing one problem to another means rephrasing the problem so it looks like a problem you can solve. This applies to the whole process of modeling: taking a real-life problem, and abstracting it so it can be phrased in the language of

¹Except for one small village, of course.

mathematics (as a linear or a quadratic equation, for example; or looking at the 14/15 puzzle as a problem on permutations), as well as to within mathematics and computer science itself (reducing the permutation problem in the 14/15 puzzle to a question of parity). The more you know about mathematics and computer science, and the better the tools you know, the more there is you can reduce to.

E.5 Negation

This is a particularly useful piece of advice when trying to show that something of the type “for all x : $P(x)$ ” is true. Proving a statement $P(x)$ for arbitrary x can be difficult. If, on the other hand, we assume the statement false, that is, we assume that there is an x such that $P(x)$ is false, then we have an x we can work with, and try to show that such an x does not exist. In SQL, this piece of advice often comes in handy when using `NOT EXISTS` to find solutions to queries that requires a property to be true for a whole set of objects (e.g. “find classes that only COMP-SCI students have enrolled in”, in other words: every student in such a class has to be a COMP-SCI student; there is no `ALL` operator in SQL², so we have to use a double negation to capture `ALL`).

E.6 Special cases, particular versions

There are two ways to read this piece of advice profitably: if your problem is parameterized, that is, has a lot of free variables, fix them. Preferably to some small or typical values. Working on these specialized versions will give you a feel for the more general problem.

Example E.6.1. Let us write $a|b$ is the integer a divides the integer b . For example $3|6$ and $7|21$, but $8|12$ is false. Suppose we were asked to determine whether it is always true that if $a|bc$, then either $a|b$ or $a|c$. After trying some small examples, a manifests itself as the interesting parameter, so let us try some small values, such as $a = 1, 2, 3$. Since $a = 1$ divides every number, this case won’t lead to a counterexample. Also, if $2|bc$, this means that bc is even, but then either b or c has to be even, that is $2|b$ or $2|c$. In other words, the statement is true for $a = 2$, and, as it turns out, for $a = 3$ as well.

At this point we need to look a bit more closely: our question really amounts to asking whether a number, a , can be split up across two factors. This, of course, can’t happen with $a = 2, 3$ since both of these numbers are prime. But what happens if we choose a to be composite, e.g. $a = 4$? Immediately a counterexample presents itself: $a = 4$ and $b = c = 2$ shows that the statement is wrong.

The other way of reading this advice is to simplify the problem by changing one of the fixed parameters. A typical example would be the modified chess-

²Well, there is, in the standard, but nobody implements it.

board problem at the beginning of the puzzle section. Instead of looking at the 8×8 board, why not consider a 4×4 or even a 3×3 or a 2×2 board instead? Studying these do not give you the answer you are looking for, but they might very well give you the clue you need to break the problem. (Have you tried a $2 \times 2 \times 2$ Rubik's Cube? Or a $4 \times 4 \times 4$ one?)

Appendix F

Set Theory

A set is the collection of its elements. We write $x \in A$ or $x \notin A$ to denote that x belongs or does not belong to A . The basic operations on sets are *union*, *intersection*, *complement* and *difference* defined as follows:

$$A \cup B = \{e : e \in A \vee e \in B\}.$$

$$A \cap B = \{e : e \in A \wedge e \in B\}.$$

$$A - B = \{e : e \in A \wedge e \notin B\}.$$

$$\bar{A} = \{e : e \notin A\}.$$

Sometimes, the symmetric difference is useful: $A \Delta B = (A - B) \cup (B - A)$. All of these operations can be easily visualized in a Venn diagram.

The notions of union and intersection can also be extended to more than two sets: we write

$$\bigcup_{i \in I} U_i = \{x : \text{there is an } i \in I \text{ such that } x \in U_i\},$$

for the union of a collection $U_i, i \in I$ of sets, and, similarly,

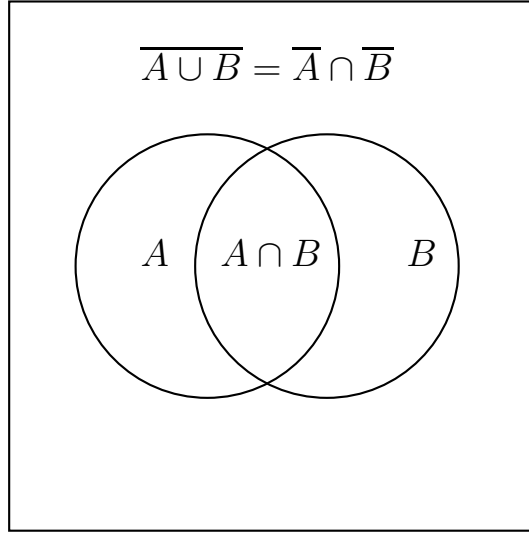
$$\bigcap_{i \in I} U_i = \{x : x \in U_i \text{ for all } i \in I\},$$

for the intersection of a collection $U_i, i \in I$ of sets.

A *partition* of a set U is a collection of sets $U_i, i \in I$ which are *pairwise disjoint*, that is, $U_i \cap U_j = \emptyset$ for all $i \neq j, i, j \in I$ and whose union is U , that is,

$$U = \bigcup_{i \in I} U_i.$$

The *cardinality* or *size* of a set A is denoted by $|A|$. For finite sets this is the number of elements in the set. The *empty set* is the set not containing any

Figure F.1: Venn diagram for A and B with complements.

elements and is denoted by \emptyset or $\{\}$. The *powerset* of a set is the set of all subsets of that set, defined as follows:

$$\mathbf{P}(A) = \{X : X \subseteq A\}.$$

For finite sets $|\mathbf{P}(A)| = 2^{|A|}$.

Many of the logical equivalences we saw in propositional logic translate naturally into set equalities. Including, for example the following:

$$\begin{array}{ll}
 A = \overline{\overline{A}} & \text{(Double complementation)} \\
 A \cap \overline{A} = \emptyset \\
 A \cap \emptyset = \emptyset \\
 A \cup \emptyset = A \\
 A \cap A = A \\
 A \cup A = A \\
 (A \cup B) \cup C = A \cup (B \cup C) & \text{(Associativity of } \cup) \\
 (A \cap B) \cap C = A \cap (B \cap C) & \text{(Associativity of } \cap) \\
 A \cup B = B \cup A & \text{(Commutativity of } \cup) \\
 A \cap B = B \cap A & \text{(Commutativity of } \cap) \\
 \overline{A \cap B} = \overline{A} \cup \overline{B} & \text{(DeMorgan)} \\
 \overline{A \cup B} = \overline{A} \cap \overline{B} & \text{(DeMorgan)} \\
 A \cap (B \cup C) = (A \cap B) \cup (A \cap C) & \text{(Distributivity of } \cap \text{ over } \cup) \\
 A \cup (B \cap C) = (A \cup B) \cap (A \cup C) & \text{(Distributivity of } \cup \text{ over } \cap)
 \end{array}$$

There are many sets that have common names, in particular in mathematics,

and in particular sets of numbers:

- $\mathbb{N} = \{0, 1, 2, 3, \dots\}$, the set of *natural numbers*. (Sometimes, 0 is not included in this set.)
- $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$, the set of *integers*.
- $\mathbb{Q} = \{x/y : x \in \mathbb{N} \wedge y \in \mathbb{N} - \{0\}\}$, the set of *rational numbers*.
- \mathbb{R} , the set of *real numbers*.
- $\mathbb{C} = \{a + bi : a \in \mathbb{R}, b \in \mathbb{R}\}$, the set of *complex numbers*.

Appendix G

First-Order Logic

First order logic is based on relations, typically written R, S, T , and so on. Depending on the number of arguments we distinguish unary $R(x)$, binary $R(x, y)$, ternary $R(x, y, z)$ and higher-arity relations $R(x, y, z, \dots)$.

A binary relation R is

- *reflexive* if $R(x, x)$ for all x .
- *anti-reflexive* if $\overline{R(x, x)}$ for all x
- *symmetric* if $R(x, y) \rightarrow R(y, x)$ for all x and y .
- *anti-symmetric* if $R(x, y) \wedge R(y, x) \rightarrow x = y$.
- *transitive* if $R(x, y) \wedge R(y, z) \rightarrow R(x, z)$ for all x, y and z .

A binary relation \equiv is an *equivalence relation* if it is reflexive, symmetric and transitive. An *equivalence class* is a set of elements equivalent to some element, it can be written as $[a]_{\equiv}$, where a is any element of the equivalence class.

A binary relation \preceq is an *ordering relation* if it is reflexive, anti-symmetric and transitive. Two elements x, y in an ordering are *comparable* if either $x \preceq y$ or $y \preceq x$. If any two elements in an ordering are comparable, the ordering is *total* otherwise it is *partial*. If instead of requiring reflexivity we ask for anti-reflexivity we obtain a *strict ordering*. Totality and partiality is defined as above.

First-order logic has two types of quantifiers: \forall , the *universal quantifier*, and \exists , the *existential quantifier*. We write

$$(\exists x)[P(x)]$$

to express that $P(x)$ is true for some value of x . Similarly,

$$(\forall x)[P(x)]$$

states that $P(x)$ is true for all values of x .

Here are some basic facts about quantifiers:

$$(\forall x)[P(x) \wedge Q(x)] \leftrightarrow (\forall x)[P(x)] \wedge (\forall x)[Q(x)],$$

that is \forall distributes over \wedge . The same is true for the existential quantifier and \vee :

$$(\exists x)[P(x) \vee Q(x)] \leftrightarrow (\exists x)[P(x)] \vee (\exists x)[Q(x)].$$

The two quantifiers are closely linked:

$$(\forall x)[P(x)] \leftrightarrow \overline{(\exists x)[\overline{P(x)}]},$$

which says that a property P is always true if and only if there is no counterexample, that is no x that makes it false. The dual of this is:

$$(\exists x)[P(x)] \leftrightarrow \overline{(\forall x)[\overline{P(x)}]},$$

which says that there is an x that makes P true if and only if P is not false for all x . These are equivalents of DeMorgan's laws for quantifiers.

A binary relation R is called *single-valued* if

$$(\forall x)(\forall y)(\forall z)[R(x, y) \wedge R(x, z) \rightarrow y = z].$$

It is called *total* if

$$(\forall x)(\exists y)[R(x, y)].$$

A *function* $f : X \rightarrow Y$ is a binary relation on $f \subseteq X \times Y$ which is total and single-valued. We call X the *domain* of f and Y the *range*.

We also write $f(X) = \{f(x) : x \in X\}$, the *image* of f . If $f(X) = Y$, we call f *onto* or *surjective*. If $f(x) = f(y)$ implies that $x = y$ for all $x, y \in X$, then f is *one-to-one* or *injective*. A function which is both onto and one-to-one is called *bijective* or a *bijection*.

Appendix H

Algorithmic Notation

We use variables and arithmetic as we would in regular mathematics, the only difference being that we write out multiplication, i.e. we would write $x * y$ and not xy . Assignment is written as $:=$. E.g.

```
input x
y := x*x
return y
```

asks the user for an input which is stored in variable x , then computes the square of x , stores it in y and then returns the value of y . We use $x \% y$ for the remainder after dividing x by y .

We add comments by using :

```
input x      // get user input
y := x*x    // compute square
return y    // return square
```

We test conditions using the `if` statement.

```
input year
if (4 | year)      // simplified
    return "leap year" // leap year rule
```

Note that we indent the code that is to be performed in case the condition succeeds. We can also specify an action in case of the failure of a condition:

```
input year
if (4 | year)      // simplified
    return "leap year" // leap year rule
else
    return "not leap year"
```

Again, the indentation of the code determines what code is performed depending on whether the condition succeeds or fails.

Complex conditions are written using Boolean operations "and", "or" and "not".

We can repeatedly perform a piece of code by making it part of a loop.

```
for i = 1 to n
  if R[i].LastName = "Johnson"
    return i
return 0 // not found
```

Here is another example:

```
input n
sum := 0
for i = 1 to n
  sum := sum + i
return sum
```

What does this program compute?

Sometimes there is a condition controlling whether we want to repeat some code or not. For that we use the `while` loop, which performs a piece of code, as long as a condition is true. For example, we can rewrite linear search

```
i := 1
while i <= n
  if R[i].LastName = "Johnson"
    return i
  i := i + 1 // go to next record
return 0 // not found
```

As long as the value of i is within bounds (at most n), the algorithm will test the last name of the current record; if it doesn't find the name we are looking for, we increase i by one and keep looking.

Appendix I

Graph Theory

A *graph* is a collection of *vertices* (or *node*) some of which are connected by *edges*. More formally, a graph G is a pair $G = (V, E)$ of vertices V and edges E , where each edge in E is a set of two vertices. For vertices we typically use letters such as u, v, s, t, w and for edges e, f, g . If e is the edge from u to v then, formally, $e = \{u, v\}$ but we will typically write $e = uv$ for simplicity. The vertices u and v are the *ends* of the edge uv , and we say that u and v are *incident* to the edge uv . The *neighborhood* of a vertex u of the graph is

$$N(u) := \{v : uv \in E\}.$$

A vertex with an empty neighborhood is called *isolated*.

A *walk* is a sequence $u_1, e_1, u_2, e_2, \dots, e_{n-1}, u_n$ of vertices and edges that traverses the graph; i.e. if you start at u_1 , e_1 takes you to u_2 , etc. In other words, $e_1 = u_1u_2$, $e_2 = u_2u_3$, and so on up to $e_{n-1} = u_{n-1}u_n$. The first and last vertices in a walk are called its *endpoints*.

There are two special types of walks: A *path* is a trail in which every vertex occurs at most once, and a *cycle* is a trail with $n \geq 3$ whose first and last vertex are the same, $u_1 = u_n$, but there are no other vertex repetitions. (We exclude the case $n = 2$ as a cycle, since it simply means we walk back and forth along the same edge.)

A graph is *connected* if there is a path between any two vertices of the graph, that is for any two vertices u and v of the graph there is a path that has u and v as endpoints. A graph that is not connected is known as *disconnected*.

$H = (U, F)$ is a *subgraph* of $G = (V, E)$ if $U \subseteq V$ and $F \subseteq E$. Two graphs are *isomorphic* if they are the same graph up to renaming the vertices. We also say $H = (U, F)$ is a *subgraph* of $G = (V, E)$ if H is isomorphic to a subgraph of G .

A *directed graph* (or *digraph*) $G = (V, E)$ consists of a set of vertices V and a set of edges between vertices, where an edge is a pair of vertices: $E \subseteq V \times V$. The edge (u, v) differs from the edge (v, u) (unless $u = v$, but we do not allow

an edge from a vertex to the same vertex). We will write uv for (u, v) for the directed edge from u to v . We draw a directed edge uv with an arrow pointing from u to v ; u is sometimes called the *child* vertex and v the *parent* vertex, in particular if we are traversing the edge in the direction of the arrow: we go from a child to a parent.

A *subgraph* of a directed graph is defined as in the undirected case except that we now remember orientation of edges.

A *directed path* is a path in a directed graph all of whose edges are oriented the same way along the path. A *directed cycle* is cycle in a directed graph all of whose edges are oriented the same way along the cycle. A directed graph is *strongly connected* if for every pair (u, v) of vertices there is a directed path leading from u to v . A directed graph is a *dag* or *acyclic* if it does not contain a directed cycle as a subgraph.

A graph $G = (V, E)$ is *bipartite* if there are disjoint sets V_1 and V_2 such that $V = V_1 \cup V_2$ and all edges in E are between vertices from V_1 and V_2 . (That is, $E \subseteq \{uv : u \in V_1, v \in V_2\}$).

A graph is *connected* if for every pair of vertices there is a path connecting the two vertices (that is, having the two vertices as endpoints).

A graph is a *tree* if it is connected and does not contain any cycles.

There are many graphs important enough to deserve names; P_n is the path on n vertices, that is $n - 1$ edges, and therefore of length $n - 1$. C_n is the cycle on n vertices (and n edges). K_n is the *complete* graph on n vertices that is, a graph with n vertices and an edge between all pairs of vertices. Its complement $\overline{K_n}$ is the *empty* graph consisting of n isolated vertices. $K_{m,n}$ is the complete bipartite graph on sets of m and n vertices with all edges between the two sets and no edges within the sets.