CSC 202 Mathematics for Computer Science
Lecture Notes

Marcus Schaefer
DePaul University[1]

Part I

Databases and Logic

Chapter 1

Databases and Propositional Logic

1.1 A Simple Database

Even if you have never seen a database before, you probably will not find the data in the following table hard to read.

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

Example 1.1.1. Jennifer Snowdon is a graduate student in computer science, living in Springfield. She started in 2004; her student SID is 93321 and her social security number is 321321321.

A *table* consists of multiple rows, also known as *records*. Each row records a point of data. The columns of the table correspond to the different *fields* or *attributes* of a record, that is, the properties or characteristics of the record that are being stored. A *relational database* allows us to store and organize many tables that contain related information.

Remark 1.1.2. For this course I suggest you use the H2 database (see Appendix A). You can also use Microsoft Access or MySQL, but these do not handle all of the examples we will see.

Data in a database is typically accessed using SQL (Structured Query Language). A simple SQL *select query* consists of three clauses, the `SELECT` clause, specifying what information we want to retrieve, the `FROM` clause, specifying which table or tables we want to retrieve the information from, and the `WHERE` clause, in which we can restrict what information we want to see. For example,

```
SELECT *
FROM Student;
```

will return *all* the data in the Student table. We can also select particular fields of the table. For example,

```
SELECT LastName
FROM Student;
```

will return the last names of students. If we run it on the Student table above, we get:

LastName
Brennigan
Patel
Snowdon
Winters
Starck
Patel
Snowdon
Johnson

Depending on whether the table contains only current students or all students, the query returns the last names of current students (or all students). Note that SQL does not remove duplicates in its output, so if two students have the same last name, that last name will occur twice in the output table (this happens twice in the example above). To remove duplicates, you can use `SELECT DISTINCT` in place of `SELECT`. For example, try running the following query:

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

We can also select several attributes at once:

```
SELECT SID, LastName, FirstName, SSN, Career
FROM Student;
```

gives us a short list of student names, social security numbers and the degree they enrolled for.

Exercise 1.1.3. What are the outputs of the following queries? (Compute the output tables, and try to interpret the data.)

```
SELECT Career
FROM Student;
```

```
SELECT LastName, SSN
FROM Student;
```

```
SELECT SSN
FROM Student;
```

As we mentioned earlier, a select query can contain a `WHERE` clause with which we can limit the output. For example,

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

will list all undergraduate student information.

Exercise 1.1.4. Write queries for the following tasks:

1. (S) List the social security numbers of all graduate students.

2. List the first names of all students whose last name is 'Patel'.

3. (S) List complete name and SID of each student who is in computer science.

4. List complete name and SID of each student who started *before* 2004.

If we were asked for a list of all students, what attributes should we select? Just the last name is a bad idea, but even first and last name is dangerous: there could be two students with the same last and first name. One more try: how about selecting last name and social security number (that number should be unique). That idea also fails for our student database, because we have two foreign students without a social security.

What we are looking for is called a *key*: a set of attributes in the table that uniquely identifies a single row, does not contain null values, and which does not contain any superfluous attributes. We saw that for the student table LastName and FirstName would not make a good key; similarly, SSN is not good, either (while it is unique if it exists, it does not always exist and null values are forbidden in key fields). If a table does not contain a good natural key, then the best solution is to introduce a key; for example, our fictitious Chicago university issues student ID numbers to every student that are promised to be unique (and every student does have an ID number). So whenever we write a query listing students, we should include the SID field, so we can really distinguish different students.

Note that while SID is a key for the student table, SID and LastName is not: they do uniquely identify a row, but LastName is superfluous, SID by itself already identifies a row.

Remark 1.1.5. Declaring a set of attributes the key of a table—known as the *primary key*—is a design decision. It is a restriction you voluntarily base on your table: the key has to be unique and its values have to be different from `null`. The database will not allow you to enter a record violating these conditions.

1.2 The Logic of Databases

The whole point of putting data into a database is so we can retrieve it later. Even better, we can now analyze the data as a whole, e.g. calculate the average salary of a company's employees, or determine which courses have high or low enrollment before scheduling them again.

Questions like these can be answered using the select statement. Let us start with a simple request: we want to list all students that are not undergraduates. If our university only has two types of students, graduates and undergraduates, then we can solve this problem as follows:

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

However, this query will stop working as soon as you add a new career option such as 'SAL' for students-at-large (students who are taking classes without working towards a degree). So it would be better to rewrite the query so it works even in that contingency. We need a way to say that `Career` is *not* 'UGRD'. Indeed, SQL allows us to write `NOT` in front of a condition to express that we want the condition to fail. The query

```
SELECT LastName, FirstName, SID
FROM Student
WHERE NOT Career = 'UGRD';
```

will list all students who are not undergraduates.[1] Let us review how SQL actually processes this select statement: first, it opens the table specified in the `FROM` clause (Student). It then looks at each row in the table, one at a time. For each row, it checks the condition in the `WHERE` clause (checking whether the career path is not undergraduate). If the condition is true, then the row is chosen, and a row containing the values in the fields selected in the `SELECT` clause are added to the output table.

Let us try another question: we want to list all undergraduate students in computer science. We have seen earlier how to list all undergraduate students, and we could certainly list all computer science students, but how do we restrict ourselves to students who are both undergraduates *and* in computer science? As SQL checks a particular record, it has to verify that both conditions are true. This is done using an `AND` to combine the two conditions:

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

[1] There is a little subtlety here: a student could return as a graduate student after finishing their undergraduate degree. That is, we would list the same student twice, once as an undergraduate, once as a graduate (we would need two records for the student, so we would have to assign two different SIDs). Should that student be listed by the query or not? One basic lesson to learn in databases is that user requests are rarely precise. Interpretation is often necessary. And dangerous.

In this query every record is checked, and only those records for which both the career *and* the program information are according to specifications are selected.

Exercise 1.2.1. Find queries to answer the following questions. Listing a student means selecting their last name, first name and SID.

1. (S) List all graduate students in computer science.

2. List all graduate students not in computer science.

3. List all undergraduate students who started before 2003.

4. (S) List all computer science students who are not from Chicago.

5. List all computer science students from Chicago who started before 2003.

6. List all students who are not from Chicago and are not in computer science.

Some of the exercises already made the point that we can have more than two conditions and that some of them might be negated. Let us do another example: we want to list all Chicago graduate students who have a social security number. That is we want to require `City = 'Chicago'` and `Career = 'GRD'` and that the student has a social security number. We don't yet know how to do the last part, but we see that the SSN field contains the value `null` for some records. To check whether the SSN field contains the null value, we write `SSN is null`. (We do *not* write `SSN = null`, which suggests that `null` is a value; on the contrary, `null` expresses the absence of a value. However, see the remark below.) Since we want the student to *have* a social security number, we need to require that `NOT SSN is null`. The whole query then becomes

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Career = 'GRD' AND City = 'Chicago' AND NOT SSN is null;
```

Note that it does not matter in what order we write the conditions; that is, we might just as well have written:

```
SELECT LastName, FirstName, SID
FROM Student
WHERE NOT SSN is null AND City = 'Chicago' AND Career = 'GRD';
```

Remark 1.2.2. The ANSI standard requires that SQL check for null values using `is null`, a feature that H2 supports. Many systems also allow `SSN = null`, which we will avoid, since it makes `null` look like just another value, which it is not. There are two ways of saying that a value is not a null: as we did above, writing `NOT SSN is null`, or using `is not null`, that is, writing `SSN is not null` in the example. Note that `is not null` is a special phrase like `is null` (which is why we used a lowercase not); for example, it is not allowed to write `SSN is not not null`, whereas we could write `NOT NOT SSN` is null. For this reason we will also avoid is not null, though this is what you will see in real queries.

One more point: Any WHERE condition we write that checks a field containing a null value through any method other than is null (or equivalent methods allowed in the system), will fail automatically. For example,

```
SELECT LastName, FirstName, SID
FROM Student
WHERE SSN = SSN;
```

will *not* return all records in the Student table, but just the ones whose SSN field contains a non-null value. We will see later when talking about foreign keys, why this convention makes sense.

Suppose somebody asked you for all the students in the computer science and the information systems degrees. What would a query for that look like? Maybe like the following query?

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

Well, no. There is not a single record that matches this query: Program cannot have two different values for the same record. The point is that the WHERE clause checks the table entries record by record. We rather carelessly misinterpreted "computer science and information systems" at the record level. The "and" in this phrase has a slightly different meaning, suggesting combining the computer science and the information systems students (it corresponds to the union of two sets as we will see in the next chapter).

Let us look at the request again: we want to list students in computer science and information systems. Looking at a particular record, we want to list that record, if Program is either 'COMP-SCI' or 'INFO-SYS'. SQL offers the OR for that:

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

This will now give us the correct answer.

Exercise 1.2.3. Find queries to answer the following questions. Listing a student means selecting their last name, first name and SID. Some might require OR some might require AND.

1. List all students from Chicago and Springfield.
2. (S) List all students that started in 2001 or 2002.
3. List all Chicago graduate students.

4. List all graduate students together with all students in computer science.
5. List all undergraduate and PhD students (a PhD student has `PROGRAM = 'PhD'`).
6. (S) List Graduate students without social security number.
7. List Graduate computer science students without social security number.
8. List all PhD students not from Chicago.

One comment about the nature of `OR`; it is an *inclusive or* that is, the query

```
SELECT LastName, FirstName, SID
FROM Student
WHERE Started = 2004 OR City = 'Chicago';
```

will list the entry for Peter Johnson (32105) who fulfills *both* conditions. An inclusive or requires that at least one, but possibly more, of its conditions be fulfilled. This is in sharp contrast to a common colloquial use of “or”, as when we say: “you can either have your cake or eat it”. It is implied that you cannot do both. Such an or is called *exclusive or*, we will encounter it again.

1.3 A Primer of Propositional Logic

Let us take a step back and look at what we have been doing: in order to increase the expressive power of select statements we developed a logic of queries which allowed us to combine conditions using operations such as `NOT`, `AND` and `OR`. The underlying structure—called *propositional logic*—is much older than databases and has applications in many other areas as well; for example, you will encounter it when writing “if” statements in programming languages. Our goal is to understand the abstract structure of propositional logic, so we can apply it to any domain we wish.

Propositional logic has an ancient ancestor in Aristotelian logic[2], but it actually is a radical departure from the viewpoint of Aristotelian logic, due to George Boole[3] (which is why it is also sometimes called Boolean logic). Boole's point of view was mathematical and algebraical rather than philosophical. In only fifty years, his approach shaped a new form of logic: mathematical logic.

Propositional logic is the logic of *propositions*, that is, statements to which we can assign a *truth-value*, namely, *true* or *false.*

Remark 1.3.1. Already, some subtle distinctions are creeping in: let us have another look at an earlier query:

[2] Aristotle lived in the third century BC. His logic is contained in six books known as the *Organon.* These books shaped logic for two millennia.

[3] Boole published his *Laws of Thought* in 1856.

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

Note that, strictly speaking, `Career = ’UGRD’` and `Program = ’COMP-SCI’` are not propositions, since they are neither true nor false by themselves. They only become true or false, when we look at a particular row in the table, say the entry for Jason Starck (`SID=19992`) and replace `Career` and `Program` with the values in that row. In the example, the condition turns into the proposition

```
’UGRD’ = ’UGRD’ AND ’INFO-SYS’ = ’COMP-SCI’
```

which is obviously false, so the record is not included in the output. On the other hand, Peter Johnson’s record (`SID = 32105`) leads to the proposition

```
’UGRD’ = ’UGRD’ AND ’COMP-SCI’ = ’COMP-SCI’
```

which is true. We will return to this distinction when talking about first-order logic.

Basic propositions can be combined by operations to form compound propositions; we have already seen three operators in the context of databases: `NOT`, `AND` and `OR`. In propositional logic, these operators are known as negation (written $\overline{\ }$), conjunction (written $\wedge$), and disjunction (written $\vee$).

In the following discussion we need to clearly distinguish two aspects of propositional logic, or any formal logic: the syntactical (how is it written?) and the semantical (what does it mean?). For example, we described how to write SQL queries correctly, so the database engine would accept them. Separately from that we also explained what the queries meant, that is, how they get interpreted. The same distinction exists in propositional logic. On the one hand, we will see how to write formulas, while on the other hand, we explain what these formulas express about propositions.

Let us take a simple example; say we have two propositions we want to argue about. We represent these by two variables p and q, also called *propositional variables*. We will talk about the proposition p and the proposition p being true or false, but, of course, p itself is not a proposition and it is not itself true or false; what we mean is that the proposition represented by p is true or false; we shorten this, confusingly, but conveniently, to “p is true or false”. (Compare this to the situation in SQL. We say that a select query lists outputs we are interested in, but it does no such thing. The query itself is just a piece of text, the database management system and the computer that process the query list the outputs.) Now the compound formula $p \wedge q$ corresponds to the condition requiring both propositions p and q to be true.

Starting with two propositions p and q, we can form the following compound formulas: $\overline{p}$, $p \wedge q$ and $p \vee q$ to denote “not p”, “p and q”, and “p or q” respectively. One important characteristic of propositional logic is that the truth-value of a compound formula only depends on the truth-values of its parts[4]. Hence we can use simple tables to explain the meaning (semantics) of the three operators, $\bar{\cdot}$, $\wedge$ and $\vee$ we have introduced.[5] To make the tables more compact we write $\bot$ in place of `false` and $\top$ in place of `true` (this is easy to remember, since $\top$ looks like a T). The variables φ and ψ are arbitrary formulas.

φ	$\overline{\varphi}$
$\bot$	$\top$
$\top$	$\bot$

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

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

Here is an example of how to read these tables: if φ is a formula we have determined to be false, and ψ is a formula we have determined to be true, then $\varphi \vee \psi$ is true, by the third line of the table for $\vee$, while $\varphi \wedge \psi$ is false, by the third line of the table for $\wedge$. Going back to our database examples, in

```
’UGRD’ = ’UGRD’ AND ’INFO-SYS’ = ’COMP-SCI’
```

φ is `’UGRD’ = ’UGRD’` and ψ is `’INFO-SYS’ = ’COMP-SCI’`. Here φ is true, and ψ is false, so $\varphi \wedge \psi$ is false, but $\varphi \vee \psi$ is true (by the third row of the truth-tables).

Nothing stops us, of course, from forming even more complex formulas. Consider, for example

$$(p \vee q) \wedge \overline{r}.$$

At this point using parentheses becomes important. Compare, for example

$$(p \vee q) \wedge \overline{r}$$

to

$$p \vee (q \wedge \overline{r}).$$

Rather like $(3 + 4) \times 2$ is not the same as $3 + (4 \times 2)$, neither are these two formulas. We can see this, by letting p be true, q false, and r true: then $p \vee q$ is true and $\overline{r}$ is false, so $(p \vee q) \wedge \overline{r}$ is false. On the other hand, $q \wedge \overline{r}$ is false, but $p \vee (q \wedge \overline{r})$ is true, since p is true. In other words, the first formula is true, and the second is false, so they cannot be the same. They actually differ on another truth assignment to p, q and r. We can find that truth assignment by systematically trying all possible truth-assignments (each proposition can be true or false, that is two possible choices; since we have three variables, we have $2 * 2 * 2 = 8$ possible assignments).

[4] Something that is not true in general. Bertrand Russell uses the example of “A believes the proposition p is true”. Whether A believes p or not, might not just depend on the truth of p (which might not even be known).

[5] History of truth-tables: Frege, Peirce, Schroeder, Wittgenstein.

p	q	r	$p \vee q$	$\overline{r}$	$(p \vee q) \wedge \overline{r}$	$q \wedge \overline{r}$	$p \vee (q \wedge \overline{r})$	
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	
$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	
$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$*$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	
$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$*$

In the table we evaluate the formula step by step by breaking it down into smaller pieces. We have marked the two rows in which the truth-value of the two formulas differ.

Exercise 1.3.2. 1. (S) Create a table with all possible truth-assignments for the formulas $\varphi = p \vee \overline{q}$ and $\psi = \overline{p \wedge q}$. Do they differ? If so, give a truth-assignment to p and q that yields different truth-values for φ and ψ.

2. Create a table with all possible truth-assignments for the formulas $\varphi = p$ and $\psi = p \vee (q \wedge \overline{p})$. Do they differ? If so, give a truth-assignment to p and q that yields different truth-values for φ and ψ.

In arithmetic, when we write

$$-3 + 4 * 5,$$

we compute the result as $(-3) + (4 * 5) = 17$ rather than $(-3 + 4) * 5 = 5$ or $-((3 + 4) * 5) = -35$ or $-(3 + (4 * 5)) = -23$. The reason are the *rules of precedence*: a negation sign binds stronger than multiplication and multiplication binds stronger than addition. In logic we have a similar rule: logical negation binds stronger than logical and which binds stronger than logical or. So when we write

$$\overline{p} \vee q \wedge \overline{r},$$

every logician would understand this to mean $(\overline{p}) \vee (q \wedge (\overline{r}))$. However, if you want to avoid confusion, add parentheses. (The same is true in SQL: when you start modifying your queries you will often wish you had added some parentheses).

Consider the formula

$$p \vee \overline{p}.$$

This formula is different from other formulas we have seen in that it cannot be made false: whatever truth-value p has, $p \vee \overline{p}$ will be true, since either p or $\overline{p}$ will be true. Formulas that are always true, regardless of the truth-values of the propositions occurring in the formula are called *tautologies*. If φ is a tautological formula, we write $\vdash \varphi$ (the notation is due to Frege), e.g. when we write

$$\vdash p \vee \overline{p},$$

we claim that $p \vee \overline{p}$ is always true, whatever the truth-value of p. The particular formula $p \vee \overline{p}$ is known as the *Law of the Excluded Middle*, since it excludes the possibility of a third truth-value: a proposition has to be either true or false. This is not very surprising in propositional logic, since we built it into the system.

At the other end from tautologies we have falsehoods: formulas that are always false. More formally, φ is a *falsehood* if

$$\vdash \overline{\varphi}.$$

Neither tautologies or falsehoods seem very useful in a `WHERE` clause: `WHERE Program = 'COMP-SCI' OR NOT PROGRAM = 'COMP-SCI'` is superfluous, since it is always true. For a condition to be interesting, there must be a way to make it true (*satisfy* it) and a way to make it false (*falsify* it). That does not mean, however, that tautologies are useless. To the contrary, they are extremely useful in rewriting and simplifying formulas.

Let us say we want to list students that are neither in computer science nor in information systems, we can write

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

Alternately, being neither in computer science nor in information systems could be phrased as saying that a student is not: in computer science or in information systems, which we could write

```
SELECT LastName, FirstName, SID
FROM Student
WHERE NOT (Program = 'COMP-SCI' OR Program = 'INFO-SYS');
```

So which query is correct? Both, since they are *equivalent*, the conditions will be true of exactly the same rows. Going back to propositional logic, we can see this as follows: we compare the truth tables of the two formulas $\overline{p} \wedge \overline{q}$ and $\overline{p \vee q}$:

p	q	$\overline{p}$	$\overline{q}$	$\mathbf{\overline{p} \wedge \overline{q}}$	$p \vee q$	$\overline{\mathbf{p} \vee \mathbf{q}}$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$

Since the truth tables of the two formulas are always the same whatever the truth-values of p and q may be, it does not matter which of the two formulas we use as far as the logic is concerned.

This rule is known as DeMorgan's law.[6] Colloquially, one could summarize it as saying that "not either" is the same as "neither". From a database point of view it often allows the simplification of a formula. Compare again

[6] Augustus DeMorgan.

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

to

```
SELECT LastName, FirstName, SID
FROM Student
WHERE NOT (Program = 'COMP-SCI' OR Program = 'INFO-SYS');
```

Imagine a large database. The first query has to evaluate the truth of each of `Program = 'COMP-SCI'` and `Program = 'INFO-SYS'` and then perform 3 operations (two `NOT` and one `AND`). The second query also has to evaluate the two basic propositions, but then performs only two operations, one `OR` and one `NOT`. So the second query will be faster than the first.

On the other hand, the first version of the query might be easier to read. A database system might present the query in one form and process it in a different form (*query optimizers* do just that: they look for equivalent queries that are faster to perform).

So it seems important that we understand the idea of logical equivalence. To define that notion formally, we introduce the equivalence operator $\leftrightarrow$ by the following truth-table:

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

We say that two formulas φ and ψ are *equivalent* if

$$\vdash \varphi \leftrightarrow \psi.$$

For example, DeMorgan's law can be stated as

$$\vdash (\overline{p} \wedge \overline{q}) \leftrightarrow \overline{p \vee q}.$$

(Note that $\leftrightarrow$ binds less strongly than $\overline{\cdot}$, $\vee$, and $\wedge$.) As a proof we can now use the earlier truth-table:

p	q	$\overline{p} \wedge \overline{q}$	$\overline{p \vee q}$	$(\overline{p} \wedge \overline{q}) \leftrightarrow \overline{p \vee q}$
$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\top$	$\top$	$\bot$	$\bot$	$\top$

Exercise 1.3.3. Using truth-tables show that $p \leftrightarrow q$ is equivalent to $(p \wedge q) \vee (\overline{p} \wedge \overline{q})$.

Here are some other equivalences that can be verified directly from the truth-tables of the formulas involved:

$$\varphi \leftrightarrow \overline{\overline{\varphi}},$$

that is, double negation is the same as assertion (i.e. no negation).

$$\varphi \wedge \varphi \leftrightarrow \varphi,$$

that is, there is no need to require a particular condition twice, since this is equivalent to requiring it once. Also,

$$\varphi \wedge \psi \leftrightarrow \psi \wedge \varphi, \text{and}$$

$$\varphi \vee \psi \leftrightarrow \psi \vee \varphi,$$

that is, with $\wedge$ and $\vee$ the order of the formulas does not matter.

There are many other basic laws of logic, many of them named. Recognizing one of them and applying it can help simplify a formula quickly.

$\varphi \leftrightarrow \overline{\overline{\varphi}}$	(Double-Negation)
$\varphi \wedge \overline{\varphi} \leftrightarrow \bot$	(Law of Contradiction)
$\varphi \vee \overline{\varphi} \leftrightarrow \top$	(Law of Exluded Middle)
$\varphi \vee \bot \leftrightarrow \varphi$	
$\varphi \vee \top \leftrightarrow \top$	
$\varphi \wedge \bot \leftrightarrow \bot$	
$\varphi \wedge \top \leftrightarrow \varphi$	
$\varphi \wedge \varphi \leftrightarrow \varphi$	(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 $\wedge$)

Exercise 1.3.4. Using truth-tables prove the laws of propositional logic above.

Remark 1.3.5. Note that all the equivalences involving $\wedge$ or $\vee$ come in two forms; any equivalence between two formulas remains true if all $\wedge$s and $\vee$s are exchanged, and every $\bot$ is switched to a $\top$ and vice versa. The reason is DeMorgan's rule, as one sees when working it out with an example (try associativity, for example).

Example 1.3.6. If you are asked to list all graduate students in computer science together with all graduate students in computer graphics, you might write the following query:

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

By distributivity, we can write the same query as

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

which is easier to read and understand: look for all graduate students in computer science and computer graphics.

Let us see how to apply the laws above to rewrite formulas. For example, we want to show that

$$\vdash (p \vee \overline{q}) \wedge (\overline{p} \vee q) \leftrightarrow (p \wedge q) \vee (\overline{p} \wedge \overline{q}).$$

We could do this using a truth-table, but now we can see why this is true more quickly:

Starting with

$$(p \vee \overline{q}) \wedge (\overline{p} \vee q),$$

we can distribute the $\wedge$ over the second formula $(\overline{p} \vee q)$, i.e. we use $\varphi = (p \vee \overline{q})$, $\psi = \overline{p}$ and $\theta = q$ and apply the distributivity of $\wedge$ over $\vee$. We get an equivalent formula

$$((p \vee \overline{q}) \wedge \overline{p}) \vee ((p \vee \overline{q}) \wedge q).$$

In both $((p \vee \overline{q}) \wedge \overline{p})$ and $((p \vee \overline{q}) \wedge q)$ we now distribute the $\wedge$ over the $\vee$ (using commutativity of $\wedge$ to argue that the order does not matter) obtaining an equivalent formula

$$((p \wedge \overline{p}) \vee (\overline{q} \wedge \overline{p})) \vee ((p \wedge q) \vee (\overline{q} \wedge q)).$$

Now $p \wedge \overline{p}$ and $(\overline{q} \wedge q)$ are both equivalent to $\perp$ by the law of contradiction, so we get

$$(\perp \vee (\overline{q} \wedge \overline{p})) \vee ((p \wedge q) \vee \perp),$$

and we can continue to

$$(\overline{q} \wedge \overline{p}) \vee (p \wedge q),$$

which, using commutativity of $\wedge$ and $\vee$, we can reorder to get

$$(p \wedge q) \vee (\overline{p} \wedge \overline{q}).$$

In summary, we showed that $(p \vee \overline{q}) \wedge (\overline{p} \vee q)$ and $(p \wedge q) \vee (\overline{p} \wedge \overline{q})$ are equivalent, i.e.

$$\vdash (p \vee \overline{q}) \wedge (\overline{p} \vee q) \leftrightarrow (p \wedge q) \vee (\overline{p} \wedge \overline{q}).$$

This last formula we proved might seem unwieldy, but that's because we did not look at it in the best possible way. The second part of the equivalence was

$$(p \wedge q) \vee (\overline{p} \wedge \overline{q}),$$

which is equivalent to $p \leftrightarrow q$. Maybe the first part can be expressed better as well: consider the piece $q \vee \overline{p}$. Either q is true, or p is false. That is, if p is true, then q has to be true as well for $q \vee \overline{p}$ to be true. In other words, $q \vee \overline{p}$ expresses *implication*, and we can define $\varphi \to \psi$ as $\overline{\varphi} \vee \psi$. Or, using truth-tables:

φ	ψ	$\varphi \to \psi$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

What does this implication do for us? If we know $\varphi \to \psi$ to be true, and we know that φ is true, then we can conclude that ψ is true. (Check that using truth-tables.) If φ is false, we cannot draw any conclusions about ψ.

Exercise 1.3.7. The version of implication, →, defined above is known as material implication. It is a rather weak type of implication, not very suitable to model stronger types of connections such as causality. Nevertheless for propositional logic it is just what we need. Consider alternative definitions, by looking at the other three possibilities for replacing the ? in the following table (the values for $\varphi = \top$ are rarely questioned) for a fictitious implication operator $\overset{*}{\to}$:

φ	ψ	$\varphi \overset{*}{\to} \psi$
$\bot$	$\bot$	?
$\bot$	$\top$	?
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Show that all possible variants either lead to operations we know or trivial operators none of which are useful in capturing implication.

Using the definition of implication, we can now give

$$\vdash (p \vee \overline{q}) \wedge (\overline{p} \vee q) \leftrightarrow (p \wedge q) \vee (\overline{p} \wedge \overline{q})$$

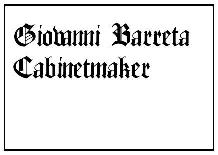
a more intuitive form; it says that

$$\vdash (p \to q) \wedge (q \to q) \leftrightarrow (p \leftrightarrow q).$$

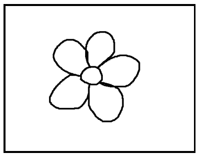
That is, two propositions are equivalent if they imply each other. Intuitively, exactly what we would want.

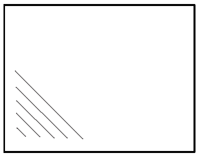
Exercise 1.3.8. Solve the following two problems.

1. You have been in the business of designing business cards for years. Your business cards always have writing (names) on the front side and graphics on the back. One rule you have stuck to is that if there is a flower motive on the back of a business card, then the writing on the front must use fancy, rather than plain, lettering. On the table in front of you are four new designs for business cards submitted by your employees (some front side up, some back side up). Determine which card(s) you have to turn around to determine whether they follow your rule about lettering:



Giovanni Barreta
Cabinetmaker





2. You are working as a bartender at a party where everybody gets issued a card which has their name and age on one side and a color code on the other side for whether they want alcoholic ("green") or non-alcoholic ("red") drinks. Since the legal drinking age in your state is 21, the guests may not have a green-backed card unless they are at least 21. After the party you are looking at some cards and you are suspicious that some might have been faked. Which ones do you need to turn around to tell which might be fake?

Angela
Bennett
Age: 20

Giovanni
Barreta
Age: 46

Red

Green

3. What is the relevant logical structure of the two exercises above? Show how this explains which cards have to be turned over.

Here are some basic rules about implication:

$\varphi \to \varphi$	
$\bot \to \varphi$	(ex falso quodlibet[7])
$(\top \to \varphi) \to \varphi$	
$(\varphi \to \bot) \to \overline{\varphi}$	(Proof by contradiction)
$(\varphi \wedge \varphi \to \psi) \to \psi$	(Modus Ponens)
$(\varphi \to \psi \wedge \psi \to \theta) \to (\varphi \to \theta)$	(Modus Barbara)
$(\varphi \wedge \psi) \to \varphi$	($\wedge$-weakening)
$\varphi \to (\varphi \vee \psi)$	($\vee$ strengthening)
$(\varphi \to \psi) \to (\overline{\psi} \to \overline{\varphi})$	(contrapositive)
$((\varphi \to \psi) \to \varphi) \to \varphi$	(Peirce's law)

[7] From falsehood, anything.

Not surprisingly, most of these rules (and names) are relevant for logical reasoning and logical proof and less useful for databases. We will talk about this at the appropriate time.

Exercise 1.3.9. Using truth-tables prove the laws of propositional logic above.

1.4 More Database Logic

A database typically consists of more than one table, and so does our sample student database. See the Appendix B for a complete listing of all tables in the database. Let us have a look at the Enrolled table. What does the second row in that table mean?

StudentID	CourseID	Quarter	Year
11035	1092	Fall	2005

The first entry is the student's ID, the second the course ID. So we can read this row as saying that the student with ID 11035 is enrolled in the course with ID 1092 in fall of 2005. After looking up the student and course information, we see that this means that Abigail Winters is enrolled for Cryptography in fall of 2005.

In database terminology `StudentID` and `CourseID` in the `Enrolled` table are *foreign keys*[8] pointing at the primary key fields of another table and identifying a particular record in that table. `StudentID` points at the primary key `SID` of `Student`, identifying a particular student, and `CourseID` points at `CID` in `Courses` identifying a particular course.

Exercise 1.4.1. 1. Interpret the other rows of the `Enrolled` table.

2. (S) What would you have to add to the database to record that Peter Johnson took the database course IT 240 in the spring of 2004?

Remark 1.4.2. This is not the place to discuss in detail why information is split into multiple tables, but we can give a quick answer. Suppose we kept all the data about students, courses, and which courses they are enrolled in, in a single table. Consider Abigail Winters. Since we have two records of her being enrolled, our single-table database would also need to have two rows for Abigail Winters, both of which repeat all of her personal information (name, city, ssn, etc.):

LN	FN	SID	SSN	Cr	Prg	City	Start	CID	CN	Dpt	Nr	Qrt	Yr
Winters	...	11035	...	GRD	PHD	Chicago	2003	1092	...	CSC	440	Fall	2005
Winters	...	11035	...	GRD	PHD	Chicago	2003	1020	...	CSC	489	Fall	2005

What's the problem with a table like this: storage space? No. Consistency! If Abigail moves anywhere else, we have to change all records which include her information. If we update inconsistently, we will have multiple Abigail Winter personas. So we place information for different objects (entities in database language) into separate tables.

[8] In case you are wondering: a foreign key is not a key by definition; it just points to a key.

In SQL we can choose multiple tables in the FROM clause of the SELECT statement. If we do so, SQL will create all possible combinations of records from the tables. We need to limit the output to the meaningful records. For example, if we wrote

```
SELECT LastName, SID, CID, CourseName
FROM Student, Enrolled, Course
```

we would get $8 * 6 * 7 = 336$ records, including many spurious ones. We only want records where the ID information between the Enrolled table and the other two tables matches. We can add this as a simple condition, similar to what we have done before:

```
SELECT LastName, SID, CID, CourseName
FROM Student, Enrolled, Course
WHERE SID = StudentID AND CourseID = CID;
```

Remark 1.4.3. The reason the student ID is called SID in Student and StudentID in Enrolled should be obvious now: if we had given them the same name, SID, say, SQL wouldn't be able to distinguish them, and we would have to write code like Student.SID = Enrolled.SID AND Course.CID = Enrolled.CID, which is always possible, and sometimes unavoidable, but cumbersome.

Here is a simple task: list all students who are enrolled in the course on theory of computation. We can start with the earlier query we saw that combines the Student, Enrolled and Course tables. Now we add the specific restriction that we want to limit ourselves to students enrolled in theory of computation. The result could look like this:

```
SELECT LastName, FirstName, SID
FROM Student, Enrolled, Course
WHERE SID = StudentID AND CourseID = CID AND
      CourseName = 'Theory of Computation';
```

Note that we did a piece of interpretation here: when we asked for all students enrolled in theory of computation, we read this to say all students enrolled in a course called theory of computation. Indeed, there are two such courses, and the client might have really meant CSC 489, in which case the query should have been

```
SELECT LastName, FirstName, SID
FROM Student, Enrolled, Course
WHERE SID = StudentID AND CourseID = CID AND
      Department = 'CSC' and CourseNr = "489";
```

Remark 1.4.4. When writing a simple database query, proceed as follows:

(*i*) Determine which tables are involved (FROM clause).

(*ii*) Determine how to connect the foreign keys in those tables with the attributes they point at (first part of the `WHERE` clause).

(*iii*) Add specific requirements (remainder of the `WHERE` clause).

How about finding all students who have *not* taken theory of computation? A first attempt could look like this:

```
SELECT LastName, SID, CID, CourseName
FROM Student, Enrolled, Course
WHERE SID = StudentID AND CourseID = CID AND
      NOT CourseName = 'Theory of Computation';
```

When you run this query, you will see that it lists (among others) Abigail Winters, although she has taken theory of computation. Why? (Think about this carefully.)

With the tools currently at our disposal, we cannot resolve the request. We will return to it in the next section.

Exercise 1.4.5. There is one query in the following set that you cannot solve yet. Figure out which one it is, and solve the others. Use `SELECT DISTINCT` when appropriate.

1. (S) List all students who are presidents of a student organization. (For this query note how the convention that conditions involving fields containing null values always fail, makes sense.)
2. List all students who have taken a course in the CSC department.
3. List students who enrolled in a class in 2006.
4. List students who were not enrolled in a class in 2006.
5. List students who have taken courses in the GPH or DC department.

SQL allows so-called *check constraints* (H2 supports these directly, in Access you can only add them through the database engine). When setting up a table you can specify that certain conditions need to be met. For example, here is part of some SQL code that could be used to set up the `Student` table.

```
CREATE TABLE student (
  LastName Text,
  ...
  CHECK (NOT SSN is null)
)
```

The check constraint at the end will be tested every time a new record is added to the table. If we try to add a student without a social security number that record is rejected (so this is probably not a good constraint). Within the

CHECK part of the CREATE TABLE statement, we can use the same conditions we use in the WHERE part of a SQL statement (the only difference is that we have to restrict the condition to the particular table we are creating). So if we wanted to require that a SSN is entered for a new student record, and the start year of the student is at least 1900 (which seems reasonable), we would write:

```
  CHECK (NOT SSN is null AND Started >= 1900)
```

Exercise 1.4.6. Write check constraints to verify the following conditions (not all of them are realistic):

1. (S) There is no course with the number '000'.
2. All courses are in the departments 'CSC', "IT", 'GPH', 'IS', or 'DC'.
3. Nobody was enrolled in Fall of 1999.
4. You cannot be a PhD student unless you are a graduate student.
5. You cannot be a PhD student unless you are a graduate student and live in Chicago.
6. All students are either 'GRD' or 'UGRD' or 'SAL' (student-at-large).
7. All graduate students have a social security number on file.
8. There are five quarters: Fall, Winter, Spring, Summer I, Summer II.

You can also add check constraints after the table has been created using the ALTER TABLE command). For example,

```
ALTER TABLE student
  ADD CONSTRAINT city_constraint
    CHECK (City in ('Chicago', 'Evanston'));
```

If you then tried to insert a new student, through

```
INSERT INTO student (LastName, FirstName, SID, City)
VALUES ('Petrarca', 'Manilo', 92200, 'Firenze');
```

this request would be rejected as a violation of the constraint city_constraint. (You do need to give a new name to every constraint you add through the ALTER TABLE command.) Compare this to

```
INSERT INTO student (LastName, FirstName, SID, City)
VALUES ('Petrarca', 'Manilo', null, 'Chicago');
```

which causes a primary key violation (SID may not be null). If you want to remove the constraint, simply say:

```
ALTER TABLE student
  DROP CONSTRAINT city_constraint;
```

1.5 More Propositional Logic

1.5.1 Truth-functions and Normal Forms

A *truth-function* is a proposition whose truth only depends on the truth of a set of propositions $p_1, p_2, \ldots, p_n$. For example, $p \vee \overline{q}$ is a truth-function of p and q. In general, we can define a truth-function through a truth-table. Consider, for example, the following table, defining the behavior of a truth-function f:

p	q	r	$f(p,q,r)$
$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\bot$

The truth-function $f(p,q,r)$ expresses that exactly one of p, q and r is true. The question we ask, is whether $f(p,q,r)$ can be expressed as a formula using propositions p, q and r. Indeed this is true, and there is a very general method to construct the formula. We do not care about the cases where $f(p,q,r)$ is false, but if it is true, we also have to make our formula true. To do this we simply take the disjunction of all those cases where the function f has to be true. In our example, there are three such cases: $p \wedge \overline{q} \wedge \overline{r}$, $\overline{p} \wedge q \wedge \overline{r}$ and $\overline{p} \wedge \overline{q} \wedge r$. So, $f(p,q,r)$ can be written as the formula:

$$(p \wedge \overline{q} \wedge \overline{r}) \vee (\overline{p} \wedge q \wedge \overline{r}) \vee (\overline{p} \wedge \overline{q} \wedge r).$$

Let us return to an earlier example: $p \leftrightarrow q$ is true if p and q have the same truth value. We saw the truth-table for logical equivalence earlier:

p	r	$p \leftrightarrow q$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Following the procedure described above, we have two cases that make $p \leftrightarrow q$ true: $\overline{p} \wedge \overline{q}$ and $p \wedge q$. Taking the disjunction of these two cases, gives us $(\overline{p} \wedge \overline{q}) \vee (p \wedge q)$, a formula we saw earlier.

Exercise 1.5.1. 1. (S) Write a formula that is true if and only if an even number of the variables p, q and r is true.

2. Write a formula that is true if and only if exactly one of the variables p, q or r is false.

3. Write a formula that is true if and only if none of p, q or r is true.

4. Write a formula that is true if and only if exactly two of p, q or r are true.

5. Write a formula that is true if p and q have opposite truth-values.

The procedure we used in the examples works in general; if $f(p_1, \ldots, p_n)$ is a truth-function of propositional variables $p_1, \ldots, p_n$, then f can be written as a formula as the disjunction of all truth-assignments which make f true.

The formulas we derive using process all have the same form; consider again our first example:

$$(p \wedge \overline{q} \wedge \overline{r}) \vee (\overline{p} \wedge q \wedge \overline{r}) \vee (\overline{p} \wedge \overline{q} \wedge r).$$

This formula is a disjunction of smaller formulas, each of which is a conjunction of propositional variables or their negation. Propositional variables and their negations are called *literals*, and a conjunction or disjunction of literals is called a *clause*. Using this terminology, we see that the formula we derived, indeed, any formula derived using the process described is a disjunction of clauses which are themselves conjunctions of literals. This way of writing a formula is known as the *disjunctive normal form (DNF)*. In other words, we have shown the following theorem:

Theorem 1.5.2. *Every truth-function can be written as a formula in disjunctive normal form.*

From this we can conclude (mathematicians call this a corollary):

Corollary 1.5.3. *Every formula is equivalent to a formula in disjunctive normal form.*

Proof. Every formula defines a truth-function, hence by the theorem it is equivalent to a formula in disjunctive normal form. ■

We already saw an instance of the corollary, when we showed how $p \leftrightarrow q$ can be written in DNF. For some formulas, the disjunctive normal form can consist of a single disjunction (e.g. $p \vee q$) or a single conjunction (e.g. $p \wedge \overline{q}$). Those are special cases of the definition.

Exercise 1.5.4.

(S) Find the disjunctive normal form of $p \wedge (q \vee \overline{r})$. (You can use either truth-tables or transformations.)

Not surprisingly, there is a dual concept, called the conjunctive normal form. A formula is in *conjunctive normal form (CNF)* if it is the conjunction of disjunctions of literals.

Corollary 1.5.5. *Every truth-function, in particular every formula, can be written in conjunctive normal form.*

Proof. Suppose we are given a truth-function or formula φ. By Theorem 1.5.2, $\overline{\varphi}$ can be written as a formula ψ in disjunctive normal form. Then $\overline{\psi}$ is in conjunctive normal form (after applications of DeMorgan to move the negation to the literal level). ■

Exercise 1.5.6. Find the conjunctive normal form of the following truth-functions.

1. $\varphi(p, q, r)$ is true if and only if exactly one of p, q, or r is true.

2. $\varphi(p, q, r)$ is true if at most two of p, q and r are true.

3. $\varphi(p, q, r)$ is true if at most one of p, q and r is true.

1.5.2 Logical Bases

Theorem 1.5.2 showed that every truth-function can be written using conjunction, disjunction and negation. There are several questions we can ask at this point: what other logical operators are there? Which of them do we actually need?

To find out what operators there are, we can see what freedom we have in defining them using truth-tables. For unary operators a truth-table looks as follows:

p	$?p$
$\bot$	?
$\top$	?

How many possibilities are there, and which of these have we seen?

More interesting is the study of binary operators:

p	q	$p \diamond q$
$\bot$	$\bot$	?
$\bot$	$\top$	?
$\top$	$\bot$	?
$\top$	$\top$	?

Exercise 1.5.7. How many ways are there to define a binary logical operator $\diamond$?

Are there any more interesting binary logical operators we have not encountered yet? Indeed, there are, for example, $p \uparrow q$, called *nand* or the *Sheffer stroke* (it is sometimes written as $p|q$, which is misleading if you use modern programming languages). It represents the case that not both p and q are true.

p	q	$p \uparrow q$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

We also have $p \downarrow q$, also known as *nor* or the *Peirce arrow*, which represents "neither p nor q".

p	q	$p \downarrow q$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$

Exercise 1.5.8. Construct the truth-table for "p unless q" and find an equivalent formula using only p, q, and any of the logical operations $\wedge$, $\vee$, and $\overline{\cdot}$.

Let us return to the second question: which of all these operators do we need in logic. Theorem 1.5.2 showed that conjunction, disjunction and negation are sufficient. Indeed, we can do without either conjunction or negation, since DeMorgan allows us to define them in terms of each other: Since $\overline{p \vee q} \leftrightarrow \overline{p} \wedge \overline{q}$, we have

$$\varphi \vee \psi \leftrightarrow \overline{\overline{\varphi} \wedge \overline{\psi}},$$

that is, we can always replace an $\vee$ in a formula using negation and conjunction. Similarly,

$$\varphi \wedge \psi \leftrightarrow \overline{\overline{\varphi} \vee \overline{\psi}},$$

so we could base logic on conjunction and negation, if we wanted to. Interestingly, a single logical operator is sufficient, as Charles Peirce observed. Indeed, either the logical nor or the logical nand are sufficient to define any of the other logical operators.

Exercise 1.5.9.

1. Use logical nands only to define negation; that is, find a formula equivalent to $\overline{p}$ only using p and $\uparrow$.
2. Use logical nors only to define negation; that is, find a formula equivalent to $\overline{p}$ only using p and $\downarrow$.
3. Show how to define nor using only nand, and nand using only nor. *Hint:* Show, and use, $\overline{p \downarrow q} \leftrightarrow \overline{p} \uparrow \overline{q}$.
4. Show how to define or using nand.
5. Show how to define and using nand.

In the exercises we saw that disjunction and negation can be expressed using the logical operator nand only. Since we already know that any truth-function can be expressed using only disjunction and negation, we have the following conclusion.

Corollary 1.5.10. *Every truth-function, in particular every formula, can be written using only the nand operator. (And the same is true for the nor operator.)*

Exercise 1.5.11. 1. Write $p \to q$ using only nand.

2. Write $p \leftrightarrow q$ using only nand.

The corollary is often expressed as saying that $\uparrow$ is a *base* for propositional logic in the sense that any truth function can be expressed using only propositional variables and the operator $\uparrow$. So we can summarize our results as saying that the following are bases:

- $\uparrow$
- $\downarrow$
- $\vee$ and $\bar{\cdot}$
- $\wedge$ and $\bar{\cdot}$

Exercise 1.5.12. Show that $\wedge$ and $\vee$ (that is, no negation) are not a base for propositional logic. *Hint:* show that formulas built solely using $\wedge$ and $\vee$ have a property called *monotonicity*: changing the assignment of a propositional variable from $\bot$ to $\top$ in the truth-table cannot turn the value of such a formula from $\top$ to $\bot$ (as negation does).

1.6 Exercises on Databases and Logic

1. Construct the truth-table for “either p or q (but not both)” and find an equivalent formula using only p, q, and any of the logical operations $\wedge$, $\vee$, and $\bar{\cdot}$. *Note:* This operator is known as “xor” or *exclusive or.*

2. A year is a *leap* year if it is divisible by 4, unless it is also divisible by 100, in which case it has to be divisible by 400 to be a leap year. Complete the following C++ code to determine whether a year is a leap year or not:

```
cin >> year; // reading input from standard input into variable year
if ( ? ) {
   cout << "year is leap year";}
else {
   cout << "year is not leap year";
}
```

   Note: In C++ you write || for $\vee$, && for $\wedge$, and ! for negation (before the condition to be negated). To test whether a number is divisible by another number use division by remainder. E.g. `year % 100 == 0` tests whether the remainder of dividing `year` by 100 is equal to 0, that is, whether `year` is divisible by 100. So, for example, `!(year % 4 == 0) && (year % 5)` returns true if `year` is divisible by 5 but not by 4. *Hint:* Recall Maass' 4th piece of advice and approach the full problem in two steps.

3. Some programming languages (functional languages in particular) contain an if-then-else expression. That is you write `if` condition `then` a `else` b, and if the condition holds, the value returned will be that of a and if not, the value of b.[9] Let's do this with propositional variables:

   (*a*) Find the truth-table of the truth-function "if p then q else r".

   (*b*) Find the disjunctive normal form for "if p then q else r".

   (*c*) Simplify the disjunctive normal form (there should be two clauses with two variables each).

4. Find the flaw in the following argument. *Hint:* the conclusion is correct and the method described was actually used, but the argument showing that the method works is wrong. *Note:* Assume an ideal rope in which knots do not shorten the rope, so the sides of the triangle really correspond to 3, 4 and 5 units.

   You probably know Pythagoras theorem: in a right triangle, the sides a, b and c relate to each other as $c^2 = a^2 + b^2$, where c is the side opposite the right angle. For example, $3^2 + 4^2 = 5^2$. Pythagoras' theorem can be practically used as follows: take 12 pieces of rope of the same length and knot them together into one long cycle. Then place the cycle on the ground so that it forms a triangle with sides consisting of 3, 4 and 5 pieces of rope (you can do that by pulling the pieces taught at the corners). By Pythagoras' theorem, the angle opposite the side with the 5 pieces of rope is a right angle which you can now use in construction.

5. (due to Martin Gardner) Multiple choice question: which of the following statements is true (assuming at least one is true)?

   (*i*) Exactly one of these statements is true.

   (*ii*) Exactly two of these statements are true.

   (*iii*) Exactly three of these statements are true.

   (*iv*) Exactly four of these statements are true.

   (*v*) Exactly five of these statements are true.

6. An old English nursery rhyme goes as follows:

   For want of a nail the shoe was lost.
   For want of a shoe the horse was lost.
   For want of a horse the rider was lost.
   For want of a rider the battle was lost.
   For want of a battle the kingdom was lost.
   And all for the want of a horseshoe nail.

[9] In C you could write $(a <= b\ ?\ a : b)$ to compute the minimum of a and b.

What is the logical structure of this nursery rhyme? Introduce appropriate atomic propositions (one for each object), and then use these propositions to describe the statements in the rhyme.

7. [Interview Question] You have three picnic baskets filled with fruit. One has apples, one has oranges, and the third has a mixture of apples and oranges. You cannot see the fruit inside the baskets. Each basket is clearly labeled. Each label is wrong. You are permitted to close your eyes and pick one fruit from one basket of your choice, then examine it. How can you determine what is in each basket?

1.7 Notes

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

Exercise 3.1.20 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.

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	$\overline{\varphi}$	$\overline{\overline{\varphi}}$	$\varphi \leftrightarrow \overline{\overline{\varphi}}$
$\bot$	$\top$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\top$

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

φ	$\varphi \wedge \varphi$	$\varphi \leftrightarrow \varphi \wedge \varphi$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

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. $(\overline{p} \wedge \overline{q} \wedge \overline{r}) \vee (p \wedge q \wedge \overline{r}) \vee (p \wedge \overline{q} \wedge r) \vee (\overline{p} \wedge q \wedge r)$.

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

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): $A \cap \overline{A} = \emptyset$.

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 3.1.8 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 3.1.11 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 8.1

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.

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

15.5 Chapter 10

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

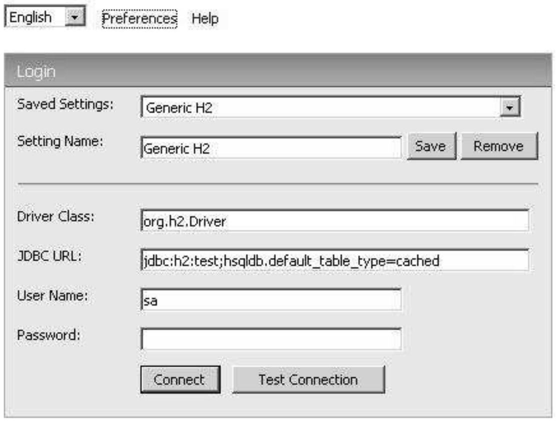
Exercise 10.0.9 $4 \not\equiv 5 \bmod 2$, since 2 does not divide $4-5 = -1$, and similarly, $4 \not\equiv 5 \bmod 3$, since 3 also does not divide $4-5 = -1$. Also, $4 \not\equiv 9088 \bmod 5$, since $4 - 9088 = -9084$ is not a multiple of 5.

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.



Saved Settings: Generic H2

Setting Name: Generic H2

Driver Class: org.h2.Driver

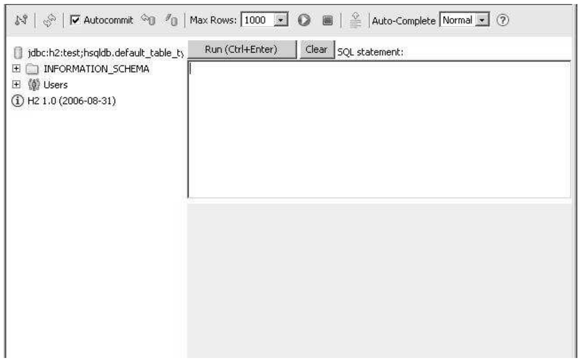
JDBC URL: jdbc:h2:test;hsqldb.default_table_type=cached

User Name: sa

Password:

Figure A.1: H2 login.

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

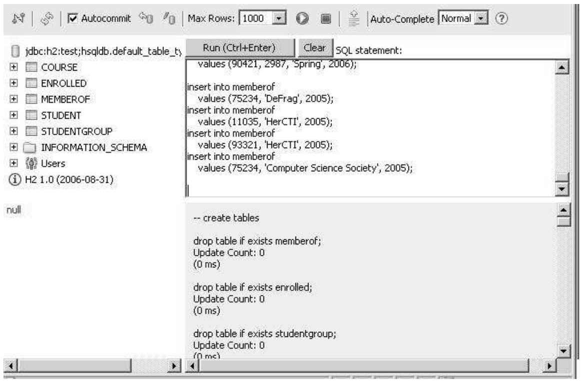


☑ Autocommit | Max Rows: 1000 | Auto-Complete Normal

jdbc:h2:test;hsqldb.default_table_ty
- INFORMATION_SCHEMA
- Users
- H2 1.0 (2006-08-31)

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



☑ Autocommit | Max Rows: 1000 | Auto-Complete Normal

jdbc:h2:test;hsqldb.default_table_ty
- COURSE
- ENROLLED
- MEMBEROF
- STUDENT
- STUDENTGROUP
- INFORMATION_SCHEMA
- Users
- H2 1.0 (2006-08-31)

null

```
  values (90421, 2987, 'Spring', 2006);

insert into memberof
  values (75234, 'DeFrag', 2005);
insert into memberof
  values (11035, 'HerCTI', 2005);
insert into memberof
  values (93321, 'HerCTI', 2005);
insert into memberof
  values (75234, 'Computer Science Society', 2005);
```

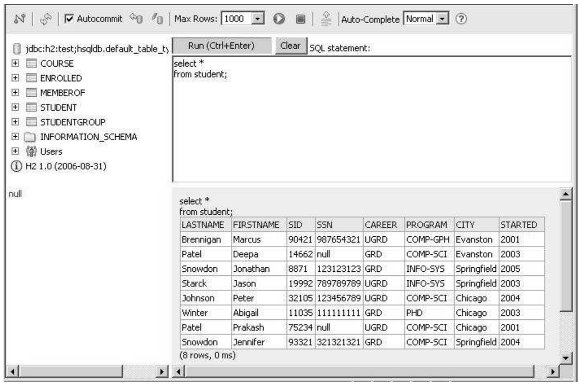
```
-- create tables

drop table if exists memberof;
Update Count: 0
(0 ms)

drop table if exists enrolled;
Update Count: 0
(0 ms)

drop table if exists studentgroup;
Update Count: 0
(0 ms)
```

Figure A.3: H2 console with database.



☑ Autocommit

Max Rows: 1000

Auto-Complete Normal

```
select *
from student;
```

```
select *
from 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	8871	123123123	GRD	INFO-SYS	Springfield	2005
Starck	Jason	19992	789789789	UGRD	INFO-SYS	Springfield	2003
Johnson	Peter	32105	123456789	UGRD	COMP-SCI	Chicago	2004
Winter	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

(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

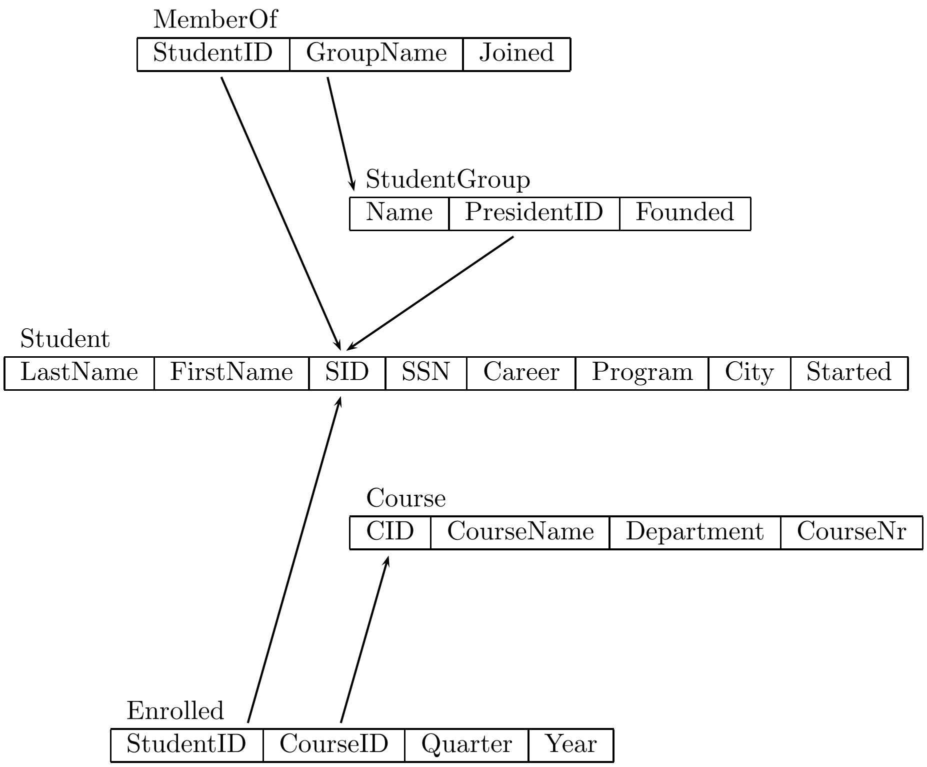


Figure B.1: Relationships in the University database

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: $\overline{\varphi}$ (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), $\varphi \to \psi$ (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	$\overline{\varphi}$
$\bot$	$\top$
$\top$	$\bot$

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

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

φ	ψ	$\varphi \to \psi$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

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

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

strongest	$\overline{\cdot}$
	$\wedge$
	$\vee$
	$\to$
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)
$\varphi \wedge \overline{\varphi} \leftrightarrow \bot$	(Law of Contradiction)
$\varphi \vee \overline{\varphi} \leftrightarrow \top$	(Law of Exluded Middle)
$\varphi \vee \bot \leftrightarrow \varphi$	
$\varphi \vee \top \leftrightarrow \top$	
$\varphi \wedge \bot \leftrightarrow \bot$	
$\varphi \wedge \top \leftrightarrow \varphi$	
$\varphi \wedge \varphi \leftrightarrow \varphi$	(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 $\wedge$)

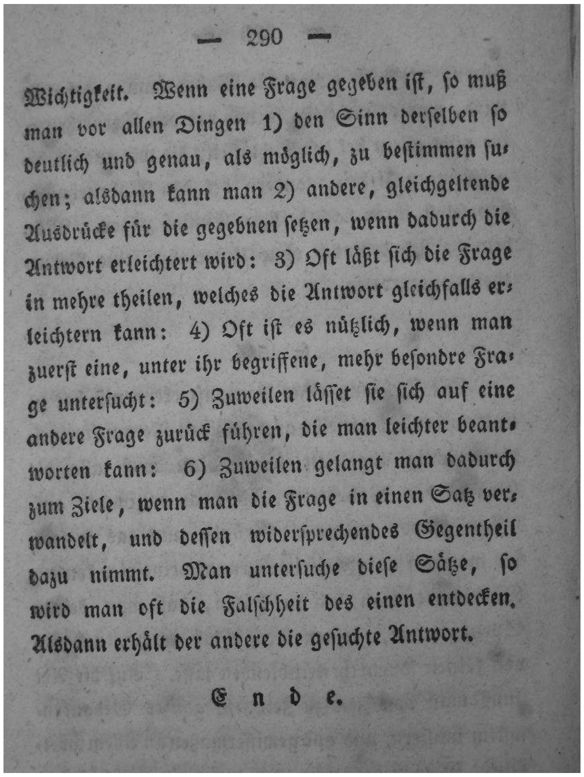
Here are some of the laws governing implication:

$\varphi \rightarrow \varphi$	
$\bot \rightarrow \varphi$	(ex falso qoudlibet[1])
$(\top \rightarrow \varphi) \rightarrow \varphi$	
$(\varphi \rightarrow \bot) \rightarrow \overline{\varphi}$	(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.



— 290 —

Wichtigkeit. Wenn eine Frage gegeben iſt, ſo muß man vor allen Dingen 1) den Sinn derſelben ſo deutlich und genau, als möglich, zu beſtimmen ſuchen; alsdann kann man 2) andere, gleichgeltende Ausdrücke für die gegebnen ſetzen, wenn dadurch die Antwort erleichtert wird: 3) Oft läßt ſich die Frage in mehre theilen, welches die Antwort gleichfalls erleichtern kann: 4) Oft iſt es nützlich, wenn man zuerſt eine, unter ihr begriffene, mehr beſondre Frage unterſucht: 5) Zuweilen läſſet ſie ſich auf eine andere Frage zurück führen, die man leichter beantworten kann: 6) Zuweilen gelangt man dadurch zum Ziele, wenn man die Frage in einen Satz verwandelt, und deſſen widerſprechendes Gegentheil dazu nimmt. Man unterſuche dieſe Sätze, ſo wird man oft die Falſchheit des einen entdecken. Alsdann erhält der andere die geſuchte Antwort.

Ende.

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 $1, \ldots, \sqrt{x}$ 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

[1] Except 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 SQL[2], 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-

[2] Well, 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:

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

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

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

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

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

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 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:

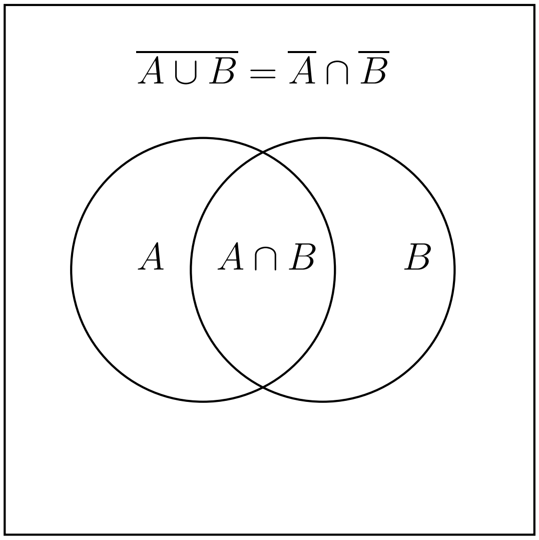


Figure F.1: Venn diagram for A and B with complements.

$A = \overline{\overline{A}}$	(Double complementation)
$A \cap \overline{A} = \emptyset$	
$A \cap \emptyset = \emptyset$	
$A \cup \emptyset = A$	
$A \cap A = A$	
$A \cup A = A$	
$(A \cup B) \cup C = A \cup (B \cup C)$	(Associativity of $\cup$)
$(A \cap B) \cap C = A \cap (B \cap C)$	(Associativity of $\cap$)
$A \cup B = B \cap A$	(Commutativity of $\cup$)
$A \cap B = B \cap A$	(Commutativity of $\cap$)
$\overline{A \cap B} = \overline{A} \cup \overline{B}$	(DeMorgan)
$\overline{A \cup B} = \overline{A} \cap \overline{B}$	(DeMorgan)
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	(Distributivity of $\cap$ over $\cup$)
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	(Distributivity of $\cup$ over $\cap$)

There are many sets that have common names, in particular in mathematics, and in particular sets of numbers:

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, the set of *natural numbers*. (Sometimes, 0 is not included in this set.)
- $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$, the set of *integers*.
- $\mathbb{Q} = \{x/y : x \in \mathbb{N} \land y \in \mathbb{N} - \{0\}\}$, the set of *rational numbers*.

- $\mathbb{R}$, the set of *real numbers.*
- $\mathbb{C} = \{a + bi : a \in RN, b \in RN\}$, 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 $R(x, y, z, \ldots)$.

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 $R(x, y) \to R(y, x)$ for all x and y.
- *anti-symmetric* if $R(x, y) \wedge R(y, x) \to x = y$.
- *transitive* if $R(x, y) \wedge R(y, z) \to R(x, z)$ 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 $u_1, e_1, u_2,\ e_2, \ldots, e_{n-1}, u_n$ 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.