COMP2111 lecture notes: sets, languages, relations, and functions.

Johannes Åman Pohjola
University of New South Wales

February 24, 2023

Contents

1 Introduction

We promised that this course would be largely self-contained: we would teach, not assume, the mathematical and logical foundations we'll need. In this document, we begin by reviewing some of the basic building blocks we'll use throughout the course: sets, languages, relations, and functions.

2 Sets

A *set* is a collection of *elements* (aka *members*). What counts as an "element" is not fixed, but can vary from application area to application area. In mathematics, we are often interested in sets where the elements are numbers. But set theory works the same for other notions of elements: we could have sets of strings, sets of programs, sets of cute baby animals, sets of literary critics, or whatever we like. We can even have sets whose elements are themselves sets.

On close inspection, the notion of set turns out to be rather slippery. (You may recall Russell's paradox from Week 1). Nailing down this slippery notion is important for studying the foundations of mathematics, and particularly so if one wants to use set theory to study itself. In this course, however, we won't do that: we are not interested in set theory for its own sake. Rather, we want to use it to model computer programs and systems. Therefore, we will not attempt to define set theory from first principles. Rather, we limit ourselves to sketching a common vocabulary we can use to define and describe sets. Nonetheless, we will attempt to develop this vocabulary in a way that allows us stay clear of paradoxes such as Russell's.

2.1 Definition by enumeration

The simplest way of defining a set is by explicitly enumerating its elements. For example, here's a set with three elements, 1, 4 and 5:

$$\{1, 4, 5\}$$

The infix symbol $\in$ is used to assert that elements are contained in sets as follows:

$1 \in \{1, 4, 5\}$	"1 is an element of $\{1, 4, 5\}$"
$4 \in \{1, 4, 5\}$	"4 is an element of $\{1, 4, 5\}$"
$17 \notin \{1, 4, 5\}$	"17 is *not* an element of $\{1, 4, 5\}$"

We snuck in an extra symbol $\notin$ for asserting that an element is *not* contained in a set. This is mainly for convenience: we can consider $a \notin S$ as syntactic sugar for $\neg a \in S$.

An important special case is the empty set, which is defined by an empty enumeration:

$$\{\}$$

It is the most boring set: it has no elements. The empty set is sometimes written $\emptyset$.

2.2 Set equality

When we define a set by enumeration, the order we write the elements in is irrelevant. For example, these sets are considered equal.

$$\{1, 4, 5\} = \{5, 4, 1\}$$

The multiplicity of the elements is also irrelevant: writing an element twice changes nothing. For example, these sets are considered equal:

$$\{1, 4, 5\} = \{1, 1, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5\}$$

The only thing that *is* relevant is which elements a set contains. In other words, a set A is uniquely defined by how the membership relation $\in$ operates on A:

Definition 1 (Set equality)**.** *Two sets A and B are considered* equal, *written $A = B$, if for all x it holds that*

$$x \in A \quad \textit{if and only if} \quad x \in B$$

There is also a standard notion of inequality on sets, called *subset* and written $\subseteq$.

Definition 2 (Subset)**.** *A set A is a* subset *of a set B, written $A \subseteq B$ if for all x it holds that*

$$x \in A \quad \textit{implies} \quad x \in B$$

For example, $\{1,2\} \subseteq \{1,2,3\}$.

2.3 Set specification

Sometimes, we're too lazy to write an explicit enumeration. Other times, we just can't. For example, the set of all natural numbers (written $\mathbb{N}$) is infinite, and we only have finite ink. Another common way of defining a set is by *specification* (aka *set comprehension*), where we implicitly define a set by specifying a property P that all its elements satisfy. They take the following general form:

$$\{\ x : P(x)\ \} \qquad \text{“the set of all } x \text{ such that } P(x) \text{ holds”}$$

Here $P(x)$ stands for some logical formula where x is a free variable. The idea is that $a \in \{\ x : P(x)\ \}$ holds precisely when $P(a)$ holds.

For example, the set $\{2,3,7,11,13,17,19\}$ could also be written as

$$\{\ n : n \in \mathbb{N} \text{ and } n \text{ is prime and } n < 20\ \} \qquad \text{“the set of all primes below 20”}$$

On avoiding paradoxes Unrestricted set comprehensions would represent a huge gaping hole in the foundations of mathematics, as first pointed out by Bertrand Russell. The issue is that we can write the following set comprehension:

$$\{\ x : x \notin x\ \} \qquad \text{“the set of all sets that do not contain themselves”}$$

For short, let $Y = \{\ x : x \notin x\ \}$. Does this innocent-looking but strangely self-referential set contain itself? Or more formally, is it true that $Y \in Y$? It turns out that if $Y \in Y$ then $Y \notin Y$, and vice versa. (Take a moment to convince yourself of this).

In other words, we have a contradiction. Contradiction is the death of logic: if your logic (or set theory, or systems modelling language) contains a contradiction, then we lose the ability to distinguish between truth and falsehood.

This suggests that set comprehensions need to be tamed somehow. What we will do is insist that all set comprehensions are of the following form:

$$S = \{\ x : x \in \mathcal{U} \text{ and } P(x)\ \} = \{\ x \in \mathcal{U} : P(x)\ \}$$

$\mathcal{U}$ here denotes a “universe”, which informally is the set of everything that is currently under consideration. For example, if we're using set theory to reason about natural numbers, the universe might be $\mathbb{N}$. Other applications demand other universes.

We will sometimes leave the universe implicit if it is irrelevant or obvious from the context. But even when it's only present in invisible ink, it serves an important purpose: it makes it so that comprehension cannot be used to conjure sets out of thin air. Instead, they must be *carved out* from some set $\mathcal{U}$ that we have already obtained by other means.

2.4 Set operators

In the same way that arithmetic comes with operators (such as $+$ and $-$) that allow us to combine numbers into new numbers, set theory has some standard operators on sets.

Union The *union* ($\cup$) of two sets includes everything that is in *either* set. Formally, $x \in A \cup B$ iff $x \in A$ *or* $x \in B$.

For example, $\{1,2\} \cup \{2,3\} = \{1,2,3\}$.

Intersection The *intersection* ($\cap$) of two sets includes everything that is in *both* sets. Formally, $x \in A \cap B$ iff $x \in A$ *and* $x \in B$.

We say that A, B are *disjoint* if $A \cap B = \emptyset$.

For example, $\{1,2\} \cap \{2,3\} = \{2\}$.

Complement The *complement* ($\cdot^C$) of a set includes everything in the universe that is *outside* the set. Formally, $x \in A^C$ iff $x \notin A$ and $x \in \mathcal{U}$.

For example, if $\mathcal{U} = \mathbb{N}$ then $\{1,2\}^C = \{0,3,4,5,6,7,8,9,10,\dots\}$.

2.5 Derived operators

Here are two common derived operators:

Set difference The *difference* ($\setminus$) of two sets A and B is obtained by removing all elements of B from A. Formally, $x \in A \setminus B$ iff $x \in A$ and $x \notin B$.

For example, $\{1,2\} \setminus \{2,3\} = \{1\}$.

Symmetric difference The *symmetric difference* of two sets includes everything that occurs in *exactly one* of the two sets. Formally, $x \in A \oplus B$ iff either (a) $x \in A$ and $x \notin B$, or (b) $x \notin A$ and $x \in B$.

For example, $\{1,2\} \oplus \{2,3\} = \{1,3\}$.

...derived how? We consider these derived because we can define them in terms of the existing operators as follows:

$$\begin{aligned} A \setminus B &\stackrel{\text{def}}{=} A \cap B^C \\ A \oplus B &\stackrel{\text{def}}{=} (A \setminus B) \cup (B \setminus A) \end{aligned}$$

2.6 Laws of Set Operations

The three basic set operations—union, intersection, and complement—are the building blocks of an algebra of sets, in the same way that $+, *$ are the building block of an algebra of numbers. The basic laws of this algebra are shown in Table 1.

The commutativity, associativity and distribution laws should all be familiar from the laws of arithmetic: imagine reading $+$ and $*$ instead of $\cup$ and $\cap$. Similarly, the identity law $A \cup \emptyset = A$ is analogous to $x + 0 = x$. The second identity law, $A \cap \mathcal{U} = A$, has no counterpart in arithmetic. In words, since A is part of the universe, the intersection of A and the universe is just A. The two complementation laws describe the relationship between a set and its complement: the union of a set and its complement is the universe, and the intersection of a set and its complement is empty.

All the 10 basic laws are provable from the definitions of set equality and the relevant operators. We will not prove them here, but the reader may wish to try one or two as an exercise.

Commutativity	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$
	$(A \cap B) \cap C = A \cap (B \cap C)$
Distribution	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Identity	$A \cup \emptyset = A$
	$A \cap \mathcal{U} = A$
Complementation	$A \cup (A^c) = \mathcal{U}$
	$A \cap (A^c) = \emptyset$

Table 1: Laws of Set Operations

.

Idempotence	$A \cap A = A$
	$A \cup A = A$
Double complementation	$(A^c)^c = A$
Annihilation	$A \cap \emptyset = \emptyset$
	$A \cup \mathcal{U} = \mathcal{U}$
de Morgan's Laws	$(A \cap B)^c = A^c \cup B^c$
	$(A \cup B)^c = A^c \cap B^c$

Table 2: Derived Laws of Set Operations

.

Some useful derived laws are shown in Table 2. *Derivable* here means provable using equational reasoning with just the laws from Table 1. For example, we can prove idempotence like this:

$$\begin{aligned} A &= A \cup \emptyset && \text{(Identity)} \\ &= A \cup (A \cap A^c) && \text{(Complementation)} \\ &= (A \cup A) \cap (A \cup A^c) && \text{(Distributivity)} \\ &= (A \cup A) \cap \mathcal{U} && \text{(Complementation)} \\ &= (A \cup A) && \text{(Identity)} \end{aligned}$$

Note that we never had to worry about how $=, \cup, \emptyset$ and friends are *defined* in the above proof: we only needed to know what laws they satisfy.

2.7 Duality

The attentive reader may have noticed that the laws in Table 1 come in pairs, called *duals*. For example, two laws share the name Distribution. In each pair, the laws are identical except that every $\cup$ and $\cap$ has been flipped upside down, and the $\emptyset$s and the $\mathcal{U}$s have been flipped. Isn't that curious?

This is not a coincidence! In fact, it suggest a striking general principle known as the *principle of duality*. Let's first define what a dual is:

Definition 3. *If A is a set defined using $\cap$, $\cup$, $\emptyset$ and $\mathcal{U}$, then $\mathsf{dual}(A)$ is the expression obtained by replacing $\cap$ with $\cup$ (and vice-versa) and $\emptyset$ with $\mathcal{U}$ (and vice-versa).*

For example, $\mathsf{dual}(A \cap (B \cup \emptyset)) = A \cup (B \cap \mathcal{U})$

Theorem 1 (Principle of Duality). *If you can prove $A_1 = A_2$ using the Laws of Set Operations then you can prove $\mathsf{dual}(A_1) = \mathsf{dual}(A_2)$*

This theorem holds because any such proof of $A_1 = A_2$ can be converted into a proof of $\mathsf{dual}(A_1) = \mathsf{dual}(A_2)$ by replacing each invocation of a law with an invocation of the dual law. For example, here is a proof of the other idempotence law:

$$\begin{aligned} A &= A \cap \mathcal{U} && \text{(Identity)} \\ &= A \cap (A \cup A^c) && \text{(Complementation)} \\ &= (A \cap A) \cup (A \cap A^c) && \text{(Distributivity)} \\ &= (A \cap A) \cup \varnothing && \text{(Complementation)} \\ &= (A \cap A) && \text{(Identity)} \end{aligned}$$

Compare this proof with the one from Section 2.6 to see the duality at work. Textually it is the exact same proof, except all symbols have been replaced with their dual symbols. This second derivation was only for illustrative purposes: we could save ourselves the effort by invoking the principle of duality.

2.8 Powerset

A very common construction in set theory is the *power set*. The power set of X is written $\mathrm{Pow}(X)$, and is the set of all subsets of X. For example, if $X = \{1, 2\}$ then $\mathrm{Pow}(X) = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$. Note that $\{\} \in \mathrm{Pow}(X)$ holds for any X, because the empty set is a subset of every set.

2.9 Cardinality

It is often relevant to count the number of elements in a set. The number of elements in X is called the *cardinality* of X, and written $|X|$ or (equivalently) $\#(X)$ or $\mathrm{card}(X)$.

For example, $|\{1, 2, 3\}| = 3$ and $|\varnothing| = 0$.

The cardinality of a powerset $\mathrm{Pow}(X)$ can be computed from the cardinality of X as follows:

Theorem 2. *When X is finite, $|Pow(X)| = 2^{|X|}$*

Try a few simple examples to convince yourself of this.

When X is finite, we have that $|X| \in \mathbb{N}$. The cardinality of infinite sets is a meaningful concept in set theory, but such cardinalities will obviously not be natural numbers: they will be some representation of infinity. Where things get strange is that we distinguish between different "sizes" of infinity; for example, $|\mathbb{N}| < |\mathbb{R}|$. Don't worry too much about that—it's beyond the scope of this course. We will mainly be concerned with the cardinality of finite sets.

3 Formal languages

As you probably know already, a *string* in programming is just a sequence of characters. A *formal language* is a set of strings—that's essentially all there is to it.

For example, here are two strings:

```
#include <stdio.h>
int main() {
    printf("Hello␣world!");
    return 0;
}
```

```
#include <stdio.h>>>>>
int main() {{{{{{{
    printf("Hello␣world!")
    return 0;
```

The string on the left is a member of the set of syntactically valid C programs; the string on the right is not. Formal languages is the mathematical formalism for defining sets of strings (such as the syntactically valid C programs) precisely and unambiguously.

Formal languages have many applications to system modelling and design beyond syntax, however. For example, they are useful for describing the behaviour interactive systems. We will return to this theme later in the course.

3.1 Basic notions

A formal language needs a finite, non-empty set Σ called the *alphabet*. A *word* is a finite sequence of elements from the alphabet Σ. We will use the variable names w, v to range over words.

For example, for the two code blocks above, the alphabet under consideration would be the set of ASCII characters (or perhaps UTF-8 characters). If we're considering machine code on the level of ones and zeroes, we might choose $\Sigma = \{0, 1\}$.

In programming, strings are traditionally written inside string quotes, like `"Hello"`. In formal languages, we dispose of the quotes and just write *Hello*. The empty word is written λ or ϵ.[1]

3.2 Length and concatenation

In this section, we'll use the running example alphabet $\Sigma = \{a, b\}$.

The length of a word w is written $\text{length}(w)$. It has the obvious definition. For example, $\text{length}(aaa) = 3$ and $\text{length}(\lambda) = 0$

Two words can be combined by forming their *concatenation*. This is analogous to string concatenation in programming. The traditional notation is to simply juxtapose the two words: the concatenation of w and v is written wv.

For example, if $w = abab$ and $v = bbb$, $wv = ababbbb$.

3.3 Sets of words

We write Σ^k for the set of all words of length k. Hence, for example, $aaa \in \{a, b\}^3$ but $abab \notin \{a, b\}^3$.

Two relevant special cases occur: Σ^0 (the set of all words of length 0) is just $\{\lambda\}$,[2] and Σ^1 (the set of all words of length 1) is just Σ.

Definition 4. *The set of all (finite) words, written Σ^*, is defined as follows:*

$$w \in \Sigma^* \quad \textit{iff} \quad w \in \Sigma^k \textit{ for some } k.$$

We will also write Σ^+ for the set of all non-empty (finite) words. Note that $\Sigma^+ = \Sigma - \{\lambda\}$.

Example 1 (Decimal numbers)**.** *We will use the tools at our disposal to define the set of well-formed decimal numbers to at most two places. That is, our language should include strings such as* 3.14, -3.14 *and* 3, *but not* 3.1415 *or* 3.1.4.1.5 *or* λ *or* .1.

For our alphabet, the obvious choice is $\Sigma = \{-, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Let D denote the set of decimal numbers. We define $w \in D$ to hold if and only if $w \in \Sigma^*$ and all the following hold

- *w contains at most one instance of $-$, and if it contains an instance then it is the first symbol.*

[1] This may cause confusion if your alphabet includes Greek letters.

[2] Note that $\{\}$ and $\{\lambda\}$ are different languages! The former contains no words. The latter contains one word, namely the empty word.

- *w contains at most one instance of ., and if it contains an instance then it is preceded by a symbol in* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$*, and followed by either one or two symbols in that set.*
- *w contains at least one symbol from* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Here is a more CS-flavoured example:

Example 2 (HTML documents)**.** *Take* $\Sigma = \{$*"<html>", "</html>", "<head>", "</head>", "<body>"*$, \ldots\}$*.*

The (language of) valid HTML documents *is loosely described as follows:*

1. *Starts with "<html>"*
2. *Next symbol is "<head>"*
3. *Followed by zero or more symbols from the set of HeadItems (defined elsewhere)*
4. *Followed by "</head>"*
5. *Followed by "<body>"*
6. *Followed by zero or more symbols from the set of BodyItems (defined elsewhere)*
7. *Followed by "</body>"*
8. *Followed by "</html>"*

Languages, such as HTML and C, typically have such formal descriptions of their syntax defined in standards documents, in a mix of natural language and formal notation.

This example makes an important technical point: alphabet symbols need not correspond to textual symbols, but can be more (or less) abstract. In compiler implementation, multi-character symbols such as those in Example 2 are often called *tokens*. The first step of a compiler (called *lexing*) is precisely to translate the input (a raw text string) to a word over a token alphabet. If this process fails, the compiler will usually abort with a syntax error.

3.4 Operations on languages

Languages are a special case of sets, so all the set operations apply to languages.

For example, suppose *French* and *English* denotes the set of sentences in French and English, respectivey. The language $French \cup English$ would include the sentences "Je ne suis pas fou." and "I am not crazy.". It would not, however, include "I ne is pas crazy"—this sentence is a member of neither French nor English. $French \cap English$ would include none of the above, but may contain some French phrases that have been loaned into English, such as "carte blanche".

There are two important set operations that are specific to languages.

Concatenation The concatenation of two languages X and Y is defined as

$$XY = \{xy \,:\, x \in X \text{ and } y \in Y\}$$

The idea is that a word of XY is the concatenation of a word from X and a word from Y. Thus, a sentence from *FrenchEnglish* might be "Je ne suis pas fou.I am not crazy.".

Kleene star Kleene star is used for self-concatenation. X^* is the set of words that are made up by concatenating 0 or more words in X. Thus, if *French* contains French sentences then *French** would correspond to French paragraphs. Note that we've seen the Kleene star before, when we defined the set of words over an alphabet, Σ^*.

Here are a few basic examples of how the set operations for languages work:

Example 3. *Let $A = \{aa, bb\}$ and $B = \{\lambda, c\}$ be languages over $\Sigma = \{a, b, c\}$.*

- $A \cup B = \{\lambda, c, aa, bb\}$
- $AB = \{aa, bb, aac, bbc\}$
- $AA = \{aaaa, aabb, bbaa, bbbb\}$
- $A^* = \{\lambda, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, \ldots\}$
- $B^* = \{\lambda, c, cc, ccc, cccc, \ldots\}$
- $\{\lambda\}^* = \{\lambda\}$
- $\varnothing^* = \{\lambda\}$

4 Relations

You've seen relations before. For example, $<$ ("less than") is a relation on numbers. $\subseteq$ is a relation on sets.

Relations are ubiquitous in computer science, and therefore studying relations and their properties helps with formalisation, implementation and verification of programs and systems. For example:

- Databases are collections of relations
- Common data structures (e.g. graphs) are relations
- Any ordering is a relation
- Functions, procedures and programs are relations between their inputs and outputs

Similarly to formal languages, relations can be viewed as just another special case of sets. To see how, we need to develop a few preliminary notions.

4.1 Cartesian products

Given two sets S and T, we may form their (binary) *Cartesian product* $S \times T$. The Cartesian product is a set of *pairs* of the form (s, t) such that $s \in S$ and $t \in T$. With pairs, unlike sets, the order matters: (s, t) and (t, s) are different (if $s \neq t$).

A familiar example is (x, y)-coordinates, which are elements of $\mathbb{R} \times \mathbb{R}$.

The above defines the special case of pairs. There are also triples (x, y, z), five-tuples (a, b, c, d, e), and in general n-tuples for any natural number n. This suggests the following more general construction for n-ary Cartesian products:

$$\bigtimes_{i=1}^{n} S_i \overset{\text{def}}{=} \{\ (s_1, \ldots, s_n) : s_k \in S_k, \text{ for all } k \text{ such that } 1 \leq k \leq n\ \}$$

4.2 Relations

An *n-ary relation* is a subset of a Cartesian product of n sets. In the context of relations, each of these n sets is called a *domain*.

For example, consider the relation $<$ on the natural numbers. We can think of this relation as a set, which satisfies $< \subseteq \mathbb{N} \times \mathbb{N}$. In this reading, when we write $x < y$ in ordinary mathematical prose, that is shorthand for $(x, y) \in <$. In this view, one definition of $<$ would be

$$\{\ (n, m) : n, m \in \mathbb{N} \text{ and there exists } k \in \mathbb{N} \text{ such that } n + k + 1 = m\ \}$$

When R is a binary relation, we will write $(x, y) \in R$ and $x\ R\ y$ and $R(x, y)$ interchangeably.

4.3 Image and converse

We can think of binary relations $R \subseteq S \times T$ as defining a notion of correspondence between S and T. This leads to the notion of *image*:

Definition 5 (Image of a relation)**.** *Let $R \subseteq S \times T$ be a binary relation and let $A \subseteq S$. The* image of A through *R is defined as*

$$R(A) \overset{def}{=} \{t \in T\ :\ (s, t) \in R \text{ for some } s \in A\}$$

In words, the image of a set A through a relation R is the set of everything that is related to A. For example, the image of $\{2\}$ through the $<$ relation on natural numbers is the set of all $x \in \mathbb{N}$ such that $x < 2$, namely $\{0, 1\}$.

Every binary relation has a *converse*, which is obtained by taking the relation backwards. For example, the converse of $<$ is $>$.

Definition 6 (Converse of a relation)**.** *Let $R \subseteq S \times T$ be a binary relation. The* converse *of R, written $R^{\leftarrow}$, is defined as follows:*

$$R^{\leftarrow} \overset{def}{=} \{(t, s) \in T \times S\ :\ (s, t) \in R\}$$

Observe that $(R^{\leftarrow})^{\leftarrow} = R$.

4.4 Binary relations on a single domain

Binary relations on a single domain form a common, and important, special case. Many familiar relations ($=$, $<$, $\subseteq$) fall in this bucket. We say that 'R is a relation on S' if $R \subseteq S \times S$.

Some other common "boilerplate" relations that fall into this category are the following:

The identity relation $E = \{\ (x, x) : x \in S\ \}$

The empty relation $\emptyset$

The universal relation $\mathcal{U} = S \times S$

The identity relation relates everything to itself; the empty relation relates nothing; and the universal relation relates everything. These relations are rather trivial, but they are nonetheless important for building a theory of relations, for the same reasons that 0 and 1 are important for building a theory of numbers.

4.5 Properties of binary relations

For binary relations on a single domain, there are five standard properties that you should know about. They are summarised in Table 3 and elaborated on below.

(R)	reflexive	$(x,x) \in R$	$\forall\, x \in S$
(AR)	antireflexive	$(x,x) \notin R$	$\forall\, x \in S$
(S)	symmetric	$(x,y) \in R \to (y,x) \in R$	$\forall\, x,y \in S$
(AS)	antisymmetric	$(x,y),\ (y,x) \in R \to x = y$	$\forall\, x,y \in S$
(T)	transitive	$(x,y),\ (y,z) \in R \to (x,z) \in R$	$\forall\, x,y,z \in S$

Table 3: Properties of binary relations.

Reflexivity A *reflexive* relation is one that relates every element of the domain to itself. Examples include $=$ and $\leq$.

A non-example follows here. Let $R \subseteq \mathbb{N} \times \mathbb{N}$ be defined by $R = \{(1,1),(2,2)\}$. This relation is *nonreflexive*; for example $(3,3) \notin R$. This illustrates that the domain is an important component of a relation's definition. If the domain had been just $\{1,2\}$, the relation would have been reflexive.

Antireflexivity An *antireflexive* relation is one that maps no element of the domain to itself. An example includes $<$. Note the distinction between nonreflexive and antireflexive relations: the relation R above is nonreflexive, but *not* antireflexive.

Symmetry A *symmetric* relation is a two-way relation. In terms of family relations, "is a sibling of" is symmetric (if I have a sibling, they are my siblings too) but "is an ancestor of" is not symmetric (I am not the ancestor of my ancestors). In more mathematical examples, $=$ is symmetric, but $<$ and $\leq$ are not.

A symmetric relation R satisfies the equation $R^{\leftarrow} = R$.

Antisymmetry An *antisymmetric* relation cannot have two different elements that are related both ways. $<$ and $\leq$, and $=$ are all antisymmetric. Of these, $=$ may be most surprising because it is also symmetric. The key idea is that antisymmetry only prohibits two-way relations between *different* elements. We have that if $x = y$ then $y = x$, but if so x and y are not different.

Example of relations that are not antisymmetric are the universal relation, and the "is a sibling of" relation.

Transitivity For a *transitive* relation, taking two steps through the relation doesn't add any new relationships that weren't already available by taking a single step. For example, $\leq$ is transitive: whenever we have $x \leq y$ and $y \leq z$, then clearly $x \leq z$.

A non-transitive relation is the "is one larger than" relation on $\mathbb{N}$, defined as $R = \{\ (x,y) : y = x + 1\ \}$. Here we have $1\ R\ 2$ and $2\ R\ 3$, but $1 \not R\ 3$.

4.6 Equivalence Relations and Partitions

An important special case is relations that are both reflexive, symmetric and transitive. We call such relations *equivalence relations*.

Intuitively, an equivalence relation captures the idea that related elements can be considered interchangeable for some purposes. For example, the equivalence relation

$$R = \{\ (n,m) : n \bmod 7 = m \bmod 7\ \} \subseteq \mathbb{N} \times \mathbb{N}$$

would be appropriate in a context where we are doing modular arithmetic, and are only interested in numbers modulo seven: therefore, in context 1 and 8 are interchangeable to us.

To mathematically characterise which objects are interchangeable under some equivalence relation R, we have the notion of *equivalence class*.

Definition 7 (Equivalence class)**.** *If $R \subseteq S \times S$ is an equivalence relations, then the* equivalence class $[s]$ *is defined as follows for all $s \in S$:*

$$[s] = \{ \; t \in S : tRs \; \}$$

The collection of all equivalence classes $[S]_R = \{ \; [s] : s \in S \; \}$ forms a partition of the domain:

$$S = \bigcup_{s \in S} [s]$$

Note that equivalence classes are disjoint: if $(s,t) \notin R$ then $[s] \cap [t] = \emptyset$

4.7 Partial orders

Another important special case of binary relations is partial orders.

A binary relation R is a *(non-strict) partial order* if it is reflexive, transitive and anti-symmetric. A *(strict)* partial order is anti-reflexive, transitive and anti-symmetric.

$=$ is not a partial order (because it is symmetric), $<$ is a strict partial order, and $\leq$ is a non-strict partial order.

We call these *partial* orders because they allow for some elements to be uncomparable. For example, consider the "is a prefix of" relation $R \subseteq \Sigma^* \times \Sigma^*$ defined as

$$R = \{ \; (v, w) : \text{there exists } u \in \Sigma^* \text{ such that } vu = w \; \}$$

R is a (non-strict) partial order such that some words are completely unrelated by R. We have both $ab \not R \, ba$ and $ba \not R \, ab$.

A partial order where distinct domain elements are related one way or the other (as is the case with $<$ and $\leq$ on numbers) we call a *total order*.

When $R \subseteq S \times S$ is a partial order, we refer to (S, R) as a poset (short for "partially ordered set").

Minimal and minimum elements We say that $x \in S$ A *minimal* element of a poset $(S, \preceq)$ if there is no $y \in S$ such that $y \preceq x$. Intuitively, this means that there is no element in S that is smaller than x. For example, the minimal element of $(\mathbb{N}, <)$ is 0. The poset $(\mathbb{Z}, <)$ has no minimal element because there is no smallest integer.

Because posets may have uncomparable elements, the minimal element is not unique in general. The poset $(\{a, b, c\}, \{(a, c), (b, c)\})$ has two minimal elements, a and b.

A *minimum* element, by contrast is an $x \in S$ such that $x \preceq y$ for all $y \in S$. Intuitively, everything else in the domain is below the minimum element, and if a minimum element exists, it is also minimal.

There are also the dual notions of *maximal* and *maximum* elements. A maximal element is an x such that there is no y with $x \preceq y$, and a maximum element is an x such that $y \preceq x$ for all $y \in S$.

4.8 Relational composition

It is often useful to compose binary relations with each other. Given two binary relations $R_1 \subseteq S \times T$ and $R_2 \subseteq T \times U$ with matching domains—note the two occurrences of T— we can construct their *composition*, denoted $R_1 ; R_2$.

$$R_1 ; R_2 \overset{\text{def}}{=} \{ \; (s, u) \in S \times U : \exists t. \; (s, t) \in R_1 \text{ and } (t, u) \in R_2 \; \}$$

Intuitively, $R_1 ; R_2$ means "first R_1 then R_2": it relates everything that can be reached by one hop through R_1, and then one hop through R_2.

For example, for the standard $<$ relation on the natural numbers, we can construct the self-composition $<;<$. If $<$ is less than, $<;<$ is "at least two less than".

Recalling the boilerplate relations from Section 4.4, we can now sketch out the contours of an algebra of relations. The following laws are all provable from the definitions we have given:

$$R;(S;T) = (R;S);T \qquad R;E = E;R = R \qquad R;\emptyset = \emptyset;R = \emptyset \qquad (R;S)^{\leftarrow} = S^{\leftarrow};T^{\leftarrow} \qquad E^{\leftarrow} = E$$

$$\emptyset^{\leftarrow} = \emptyset \qquad \mathcal{U}^{\leftarrow} = \mathcal{U} \qquad (R^{\leftarrow})^{\leftarrow} = R$$

5 Functions

Functions are ubiquitous in mathematics and computer science, but it's important to keep in mind that a mathematical function, and a function in a computer program, are not the same thing.

Mathematical functions, such as $f(x) = x^2 - 3$, map inputs to outputs. Let's say the f from the previous sentence is a function from $\mathbb{Z}$ to $\mathbb{Z}$. This means that for every $x \in \mathbb{Z}$, $f(x)$ *is* a value from $\mathbb{Z}$.

Now consider:

```
int f(int x) {
  return(x * x - 3);
}
```

Clearly the C function `f` implements the mathematical function f.[3] But it's not quite correct to say that `f(x)` *is* an **int**. Rather `f(x)` is an expression that, *when evaluated*, will eventually return an **int**.

Here are some examples of C functions that make the distinction even clearer:

```
double g() {
  return(Math.random());
}

int h(int x) {
  printf("Checkmate,␣maths!!");
  return x;
}
```

For mathematical functions, we always get the same value back: $f(x) = f(x)$. But `g() == g()` will not necessarily be `true`. h behaves like the identify function, except it also produces side effects.

Despite this distinction, the word function tends to be used for both notions. When we want to emphasise the distinction, functions in the programming sense are sometimes called *subroutines* or *procedures*.

5.1 Mathematical functions

Formally, a *function*, $f : S \to T$, is a binary relation $f \subseteq S \times T$ such that for all $s \in S$ there is exactly one $t \in T$ such that $(s, t) \in f$. We write $f(s)$ for the unique element related to s.

Every function has a *domain*, *co-domain* and *image*. For a function $f : S \to T$, they are defined as follows:

		Symbol	
S	*domain* of f	$\mathrm{dom}(f)$	(inputs)
T	*co-domain* of f	$\mathrm{codom}(f)$	(*possible* outputs)
$f(S)$	*image* of f	$\mathrm{im}(f)$	(*actual* outputs)
$= \{ f(x) : x \in \mathrm{dom}(f) \}$			

[3]Here we ignore the complication that there are infinitely many integers, but only finitely many of them fit in an **int**.

For example, consider the function $abs : \mathbb{Z} \to \mathbb{Z}$ that computes the absolute value of its input. It has domain $\mathbb{Z}$, co-domain $\mathbb{Z}$ and image $\mathbb{N}$.

Functions are most often presented equationally:

$$abs(x) = sgn(x) * x$$

sgn here denotes the sign function. To emphasise the point that functions are just relations with special properties, here are some equivalent alternate presentations:

$$abs = \{\ (x, sgn(x) * x) \in \mathbb{Z} \times \mathbb{Z} : T\ \} \qquad abs = \{\ (x, -x) \in \mathbb{Z} \times \mathbb{Z} : x < 0\ \} \cup \{\ (x, x) \in \mathbb{Z} \times \mathbb{Z} : x \geq 0\ \}$$

Composition and identity Function composition, written $f \circ g$, denotes a function that given an input first applies g, then applies f to the intermediate result:

$$(f \circ g)(x) \overset{\text{def}}{=} f(g(x))$$

This composition is defined whenever $\text{im}(f) \subseteq \text{dom}(g)$.

Since functions are just relations, the perceptive reader may have noticed that relational composition ; also applies to functions. In fact, when f and g are functions we have

$$f ; g = f \circ g$$

The identity function $\text{Id}_S : S \to S$ is the most boring function: it just returns its input unchanged. It may not seem like much, but it has nice algebraic properties.

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad f \circ \text{Id}_S = f \qquad \text{Id}_S \circ f = f$$

Here we subscript the identity function with its intended domain, emphasising that the identity functions for two distinct domains are formally not the same function. We will elide the domain when it is irrelevant, obvious or just in the way.

Injectivity and surjectivity The most important properties of functions are injectivity and surjectivity.

An *injective* function maps different inputs to different outputs:

$$f(x) = f(y) \to x = y$$

If a function is injective, no information is lost as a result of applying f. For example, any encryption function worth its salt had better be injective—otherwise, the same ciphertext could be obtained from different cleartexts, leaving the recipient to guess the intended message.

A *surjective* function is a function whose possible outputs cover the entire co-domain:

$$\text{im}(f) = \text{codom}(f)$$

Functions that are both injective and surjective are called *bijective*. Bijective functions have the property that they are invertible. Specifically, if $f : S \to T$ is bijective then $f^{\leftarrow}$ is a function such that $f^{\leftarrow}(f(s)) = s$ for all s.

When f is bijective, we will therefore write $f^{\leftarrow}$ as f^{-1} (pronounced "the inverse of f").

Partial functions A *partial function* $f : S \nrightarrow T$ is a binary relation $f \subseteq S \times T$ such that for all $s \in S$ there is at most one $t \in T$ such that $(s, t) \in f$. That is, it is a function $f : S' \longrightarrow T$ for some $S' \subseteq S$.

The canonical example of a partial function is division on the real numbers: its domain is $\mathbb{R} \times \mathbb{R}$, yet it is undefined when the denominator is 0.

5.2 Procedures as relations

While functions in the programming sense are different from functions in the mathematical sense, they share an important thing in common: they can both be modelled as relations between input and output. For example, consider the following function:

```
int f(int x) {
  while(x != 0) {
    x--;
  }
  return 0;
}
```

This function will return 0, except when x is negative. In the latter case, it loops forever. To capture this, we can model f as the following relation between inputs and outputs as follows:

$$\{(0,0),(1,0),(2,0),\ldots\}$$

The fact that this relation has no input-output pairs for negative inputs models the fact that the function returns no value for negative x. Here's another example, where we assume that `roll_d6()` returns a random integer between 1 and 6:

```
int nd6(int n) {
  int sum=0;
  while(n != 0) {
    sum = sum + roll_d6();
    n--;
  }
  return sum;
}
```

The intuition is that `nd6(n)` returns the result of rolling n 6-sided dice. An input-output relation R representing this function could be defined as follows:

$$R = \{\ (n,m) \in \mathbb{Z} \times \mathbb{Z} : 0 \leq n \leq m \leq 6n\ \}$$

For example, the fact that rolling 2d6 can yield e.g. 3 or 11 is modelled by the fact that both $2\ R\ 3$ and $2\ R\ 11$.

This way of modelling functions as relations deliberately abstracts from many aspects of the function's behaviour. It ignores function's runtime comlexity, and the possibility of integer overflow. It only considers what results are *possible* given an input, and completely ignores their probability.

However, this abstraction is useful: for most purposes knowing what's possible suffices to prove that a program meets its specification.

This doesn't account for the possibility that programs have side-effects. For example, if a function changes the value of a global variable before returning, this is not modelled by its input-output relations. For such programs, it is better to model programs as relations between initial and final states. More on that later in the course.

Acknowledgements

These notes borrow some from material developed by previous COMP2111 staff, including but not limited to Paul Hunter, Carroll Morgan, Liam O'Connor and Christine Rizkallah.