Logic, languages and computability

John Stalker

Contents

Introduction

In this module we'll talk about formal languages, computability and mathematical logic. Before going through each in turn it may be useful to see, in a simplified example, how closely related they are. The simplified example will be that of a module enrollment system.

A simplified example

Module enrollment systems take data from university staff about what modules students are allowed to take and from students about what modules they wish to take and either enroll the student in the modules if their choices are allowed or don't if they aren't, hopefully with some feedback about why they aren't allowed.

A real such system has to cope with many details which we'll ignore in this simplified example, like the fact that a university typically has hundreds or thousands of categories of students, depending on entry route, intended degree, year of study, etc. and that each of these groups has different restrictions on the modules they can take. All of that detail is important in a real system but in a hypothetical system intended just to illustrate some basic ideas it would just be a distraction so we'll assume here that all students have the same set of choices. We'll also ignore issues of time, such as whether a student may have taken a prerequisite module in a previous year. We'll also ignore most user interface considerations.

Rules

A very restrictive system might offer students a short list of possible combinations and ask them to pick one. An incredibly lax system might allow students to pick any combination they like. Both of these are easy to implement but any real university will have something in between and in this one way, at least, we'll try to be realistic. The usual way to specify a set of combinations is with rules, like "If you take Statistics you must also take Probability" or "You must take one and only one of these three modules". You find rules like these in a course handbook and the system's job is turn those rules and turn them into an algorithm which approves or rejects a selection.

Languages

We need to talk about languages, and the distinction between natural and formal languages. The rules above are in a natural language, specifically English, and natural languages are ambiguous. The rule "You must take Probability and Statistics or Algebra and Geometry", for example, is ambiguous in multiple ways. There is the distinction between inclusive and exclusive "or", for example. Are you allowed to take both Probability and Statistics and Algebra and Geometry or do you have to choose only one pair? How do the Boolean operators "and" and "or" split the phrase "Probability and Statistics or Algebra and Geometry" into meaningful pieces? Are there two possibilities, "Probability and Statistics" and "Algebra and Geometry", where you have to take one pair or the other? In other words, does the word "or" join separate phrases, each joined by an "and"? Or is it the other way around? In other words, do you have to take Probability, either Statistics or Algebra, and Geometry, three modules where in one case you have a choice between two? Are you allowed to take any modules beyond the ones listed? Does the phrase "Probability and Statistics" even refer to a pair of modules named "Probability" and "Statistics" or is there a single module named "Probability and Statistics"?

You may well be able to guess the intended meaning of the sentence but you're only able to do so from knowing a lot of context and you may guess wrong. Your guesses for this rule and for others will probably give the same word different meanings in different sentences. It's likely, for exam-

ple, that you interpreted the "or" in the sentence above exclusively, so that students cannot take both pairs of modules. But in a statement of prerequisites, like "Before taking Partial Differential Equations you must take Techniques in Theoretical Physics or Ordinary Differential Equations" you'd probably interpret it inclusively, so that a student who had taken both of those modules would also be allowed to take Partial Differential Equations.

To avoid ambiguities like the ones above we need formal languages. Formal languages have a precisely described grammar, which then determines how they are parsed. If you want to program to check module choices it needs them to be expressed in a formal language. Some human will then need to translate from the rules from the natural language they're expressed in a course handbook to a formal language. That formal language may look superficially like a natural language. We could, for example, continue to use "and" and "or" as logical connectives. But they'd now be used in a way which permits purely mechanical processing rather than human intuition, and they might therefore be interpreted in a way which doesn't accord with your intuition.

Statements

Rules in a course handbook are full of modal verbs like "must", "should", "may", etc. It's possible to study what's called modal logic, which attempts to formalise the meaning of such verbs. We won't do that in this module. We also wouldn't need to in order to build a module enrollment system. The part of the system which actually implements the rules is a checking procedure which takes a list of modules entered by the student and checks whether they do or don't satisfy the requirements. In describing such a procedure it's more natural to express things declaratively than imperatively. The rule which appears in course handbook as "You must take Probability and Statistics or Algebra and Geometry" can be rewritten as the statement "The student is taking Probability and Statistics or Algebra and Geometry". The checking procedure checks whether this statement, along with any others it's been given, is true for the student whose choices it's validating. I'll generally use this point of view, with statements in place of rules, from now on.

A formal language

Since we need a formal language anyway might as well dispense with everything superfluous and replace "The student is taking Probability and Statistics or Algebra and Geometry" with just "Probability and Statistics or Algebra and Geometry". There's no point in starting every single rule with "The student is taking". Our language will then consist of module names joined by the Boolean operators "and", "or" and "not" according to fixed rules.

We'll avoid the ambiguity of whether "Probability and Statistics" is one module or two by using symbols unlikely to occur in a module name to stand in for "and", "or" and "not". Specifically we'll use $\wedge$ for "and", $\vee$ for "or" and $\neg$ for "not". So "Probability and Statistics" is a single module and "Probability $\wedge$ Statistics" is the two modules "Probability" and "Statistics" joined by the logical operator $\wedge$, with we interpret as "and".

For a real online module enrollment system I would probably make another choice, for example using module codes in place of module names, but the choice above does have a few advantages. Module names are easier for humans to read and write than module codes. Also, we're less likely to confuse the ambiguous English words with the precise meanings I'll soon give to the Boolean operators. The main reason I've chosen to use $\wedge$, $\vee$ and $\neg$ though is that these are the stand symbols in mathematical logic, which will be one of the main topics of this module.

What would have appeared in the course handbook as "You must take Probability and Statistics or Algebra and Geometry" is now "Probability $\wedge$ Statistics $\vee$ Algebra $\wedge$ Geometry". We still need to resolve the ambiguity about how to split this up, which we'll do by declaring that $\neg$ takes precedence over $\wedge$, which in turn takes precedence over $\vee$. By precedence we mean that it binds more tightly, so given the choice between binding the names "Probability" and "Statistics" with an $\wedge$ or "Statistics" and "Algebra" with an $\vee$ we prefer to bind "Probability" and "Statistics" together first. Only after "Probability" and "Statistics" have been bound together with $\wedge$ and "Algebra" and "Geometry" with $\wedge$ do we bind the two larger phrases "Probability $\wedge$ Statistics" and "Algebra $\wedge$ Geometry" together with $\vee$.

While not strictly necessary, it is convenient to allow the use of parenthe-

ses to override these precedence rules. The alternative interpretation described earlier could then be written as "Probability $\wedge$ (Statistics $\vee$ Algebra) $\wedge$ Geometry". This could also expressed without parentheses as "Probability $\wedge$ Statistics $\wedge$ Geometry $\vee$ Probability $\wedge$ Algebra $\wedge$ Geometry", but this is longer and harder to read than the version with parentheses.

Note that this use of the word precedence may not match your intuitions. If you parse statements in a top down manner, which is the way humans generally do, then you start with the operators of lowest precedence and work your way up to those of higher precedence. The terminology above, which is standard, assumes a bottom up parsing, starting from the smallest units and gradually combining them until we have the full statement. Most, but not all, parsing algorithms used by computers work this way.

Logic

If the language above looks familiar, except for the role of module names, that's because it's one that's often used. With search terms in place of module names, and with the usual English names for the Boolean operators in place of the symbols $\wedge$, $\vee$ and $\neg$, it's the language used by search engines, not just the big web search engines but also the one used to search for books or articles in our library.

This formal language, with Boolean variables in place of module names, combined with various axioms and rules of inference we'll discuss later, forms what's called the predicate calculus, also known as zeroeth order logic. Beyond zeroeth order logic there is first order logic, which introduces new language elements like quantifiers, and axioms and rules of inference for them. In a real module enrollment system there would be advantages to introducing at least some elements of first order logic. For example, suppose we want to implement the simple rule "You must take Probability and Statistics and no other modules." The statement "Probability $\wedge$ Statistics" is not a faithful translation of this rule into our formal language because it doesn't enforce the "and no other modules part. The correct translation of this into our formal language would look like"Probability $\wedge$ Statistics $\wedge$ $\neg$ Algebra $\wedge$ $\neg$ Geometry ..." where the "..." continues on to list every other module offered. That's awkward. It would be much better to be able to say something like "Probability $\wedge$ Statistics $\wedge$

$\forall\ x\ .\ ((\ x = \text{Probability}\ )\ \vee\ (\ x = \text{Statistics}\ )\ (\ \vee\ \neg\ \text{x}\ ))$" where $\forall$ is the quantifier "for all" and $=$ is what you think it is. x here is a variable, which in this context is a placeholder for an arbitrary module. The disadvantage of using first order logic is that it complicates parsing input from staff, which we'll talk more about shortly, and checking input from students, which we'll talk about later. For purposes of this simple example we will therefore stick to zeroeth order logic and postpone any further discussion of first order logic until later in the module.

Most humans would not naturally write "and no other modules", thinking it obvious from context. It would then be implicit in "You must take Probability and Statistics", but might not be in other uses of the word "and". In the sentence "Before taking Forecasting you must take Probability and Statistics" it seems unlikely that there's an implicit "and no other modules". The word "and" in English therefore has at least two different interpretations, which we can usefully refer to as "exclusive 'and' " and "inclusive 'and' ". English is far from unique in failing to distinguish between these but some other languages, like Japanese, do.

Parse trees

The process described above, splitting a statement up into successively smaller phrases until we get to the simplest possible components, is called parsing. A common way to describe the result, both for natural and for formal languages, is with tree diagrams. There are two slightly different ways to do this, which are perhaps best illustrated by examples.

The first three accompanying figures give abstract syntax trees for three different parsings of "Probability $\wedge$ Statistics $\vee$ Algebra $\wedge$ Geometry". These are followed by three parse trees for the same three parsings.

A tree has elements called nodes and has arrows from one node to another. The nodes with no arrows going out are called the leaves of the tree. There is a single node with no arrows coming in, which is called the root. In the case of the syntax trees the leaves are all labelled by module names and the other nodes are all labelled by Boolean operators. In the case of the parse trees the leaves are labelled either by module names or by Boolean operators and the other nodes are unlabeled.

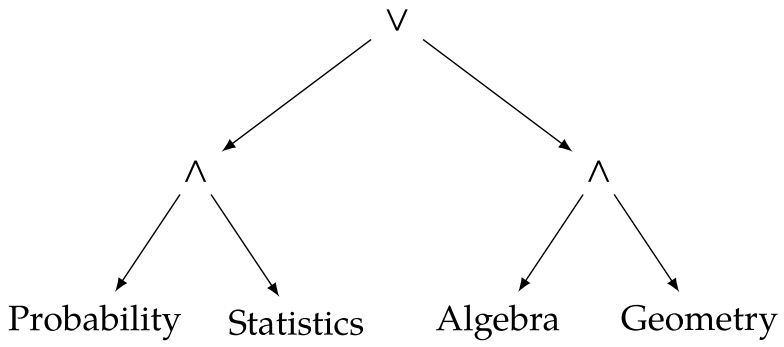


Figure 1: Syntax tree for $((\text{Probability} \wedge \text{Statistics}) \vee (\text{Algebra} \wedge \text{Geometry}))$

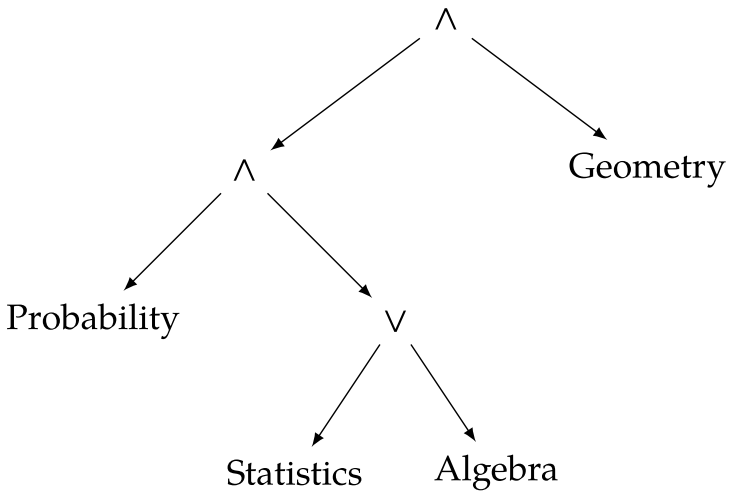


Figure 2: Syntax tree for $((\text{Probability} \wedge (\text{Statistics} \vee \text{Algebra})) \wedge \text{Geometry})$

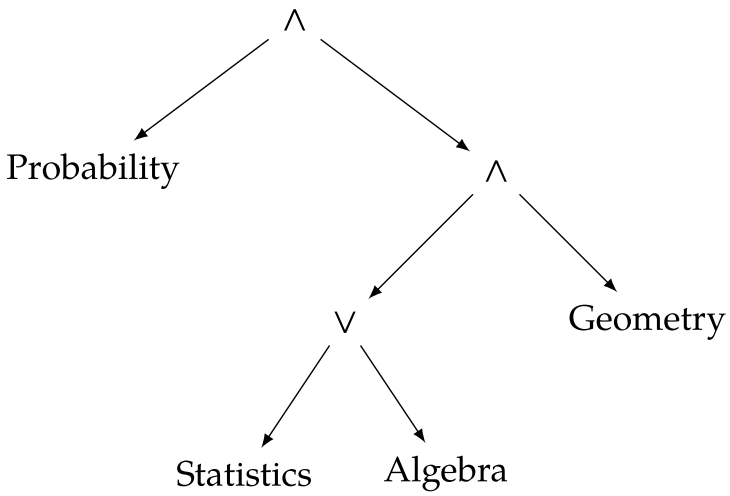


Figure 3: Syntax tree for (Probability ∧ ((Statistics ∨ Algebra) ∧ Geometry))

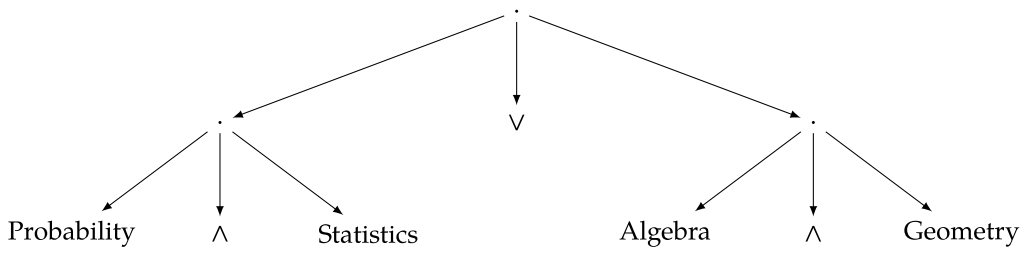


Figure 4: Parse tree for ((Probability ∧ Statistics) ∨ (Algebra ∧ Geometry))

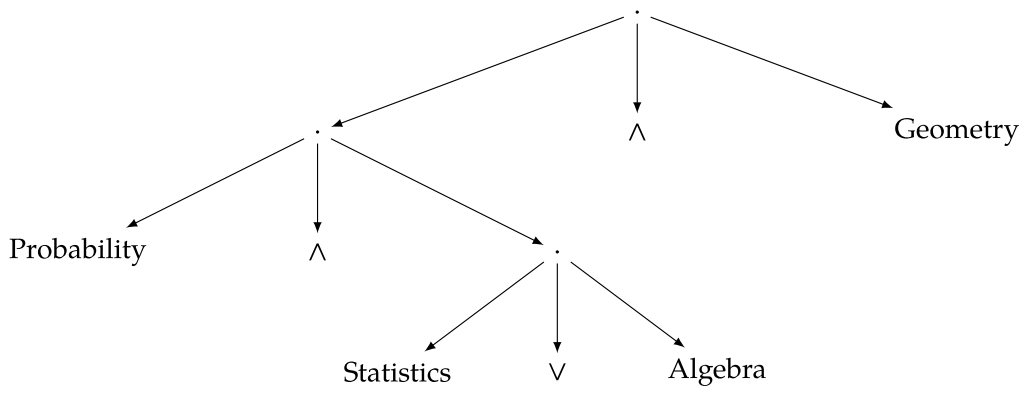


Figure 5: Parse tree for ((Probability $\wedge$ (Statistics $\vee$ Algebra)) $\wedge$ Geometry)

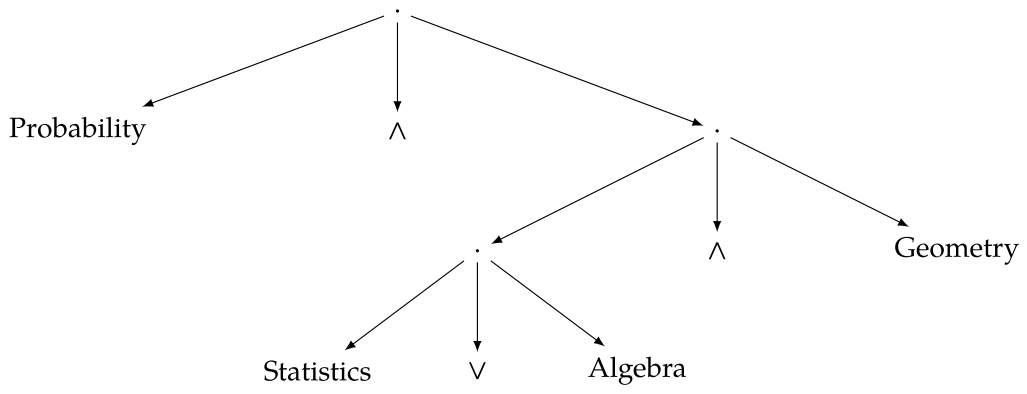


Figure 6: Parse tree for (Probability $\wedge$ ((Statistics $\vee$ Algebra) $\wedge$ Geometry))

These visual representations are nice, but all computers, and many humans, are blind. It's possible to describe the same information in a different way, with fully parenthesised expressions. The fully parenthesised expressions corresponding trees above are

```
((Probability ∧ Statistics) ∨ (Algebra ∧ Geometry))

((Probability ∧ (Statistics ∨ Algebra)) ∧ Geometry)

(Probability ∧ ((Statistics ∨ Algebra) ∧ Geometry))
```

Parse trees correspond exactly to fully parenthesised expressions. Each pair of balanced parentheses corresponds to a non-leaf node in the diagram. The internal representation a computer would use for a tree data structure isn't any of these. The visual description and the parenthesised expressions are just for humans.

The fact that there are two possible abstract syntax trees for "Probability ∧ (Statistics ∨ Algebra) ∧ Geometry", depending on which ∧ has higher precedence, shows that our grammar isn't fully unambiguous, even after specify the precedence of operators.

When parsing statements in a formal language with a program one often wants to construct a data structure which mirrors this structure. For simple enough languages though it may be possible, as we'll see, to use simpler data structures than a tree.

Graphs

A tree is a special case of a more general structure called a directed graph. A graph has nodes, which in the context of graph theory are usually called vertices, and arrows, which in this context are called edges. Note that this usage of the word "graph" has no relation at all to the graph of a function.

There are also undirected graphs, where the edges that connect vertices don't have a preferred direction.

Graphs appear in a lot of contexts, and we'll see them again in later chapters. Here I'll just give two examples, one of a directed graph and one of an undirected graph.

One common use of graphs is in understanding the structure of computer programs. The call graph of a program shows which functions call which. I wrote the following Racket program, for example, to solve Max Bezzel's classical problem of finding all configurations of 8 queens on a chessboard such that none of them can reach any of the others in a single move.

```
#lang racket
(require srfi/1)
(define (id x) x)
(define (curry f arg1) (lambda (arg2) (apply f (list arg1 arg2))))
(define (rcurry f arg2) (lambda (arg1) (apply f (list arg1 arg2))))
(define (interval a b) (unfold (curry < b) id (curry + 1) a))
(define (shift slope s)
  (if (null? s)
      '()
     (cons (car s) (map (rcurry + slope) (shift slope (cdr s))))))
(define (duplicates? s)
 (any (lambda (t) (member (car t) (cdr t))) (unfold null? id cdr s)))
(define (okay? slope s) (not (duplicates? (shift slope s))))
(define (queens n)
  (filter (curry okay? 1)
          (filter (curry okay? -1)
                  (permutations (interval 1 n)))))
```

This in fact solves the problem for a chessboard of arbitrary size, not just the classical one of size eight. I'm not going to explain how it works, but there are eight functions defined and the first step towards understanding how this works would be to see which functions call which other functions, which is described by the call graph in the figure.

For an example of an undirected graph we consider a different classical puzzle, first posed by Alcuin of York:

> Homo quidam debebat ultra fluvium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam navem invenire, nisi quae duos tantum ex ipsis ferre valebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?

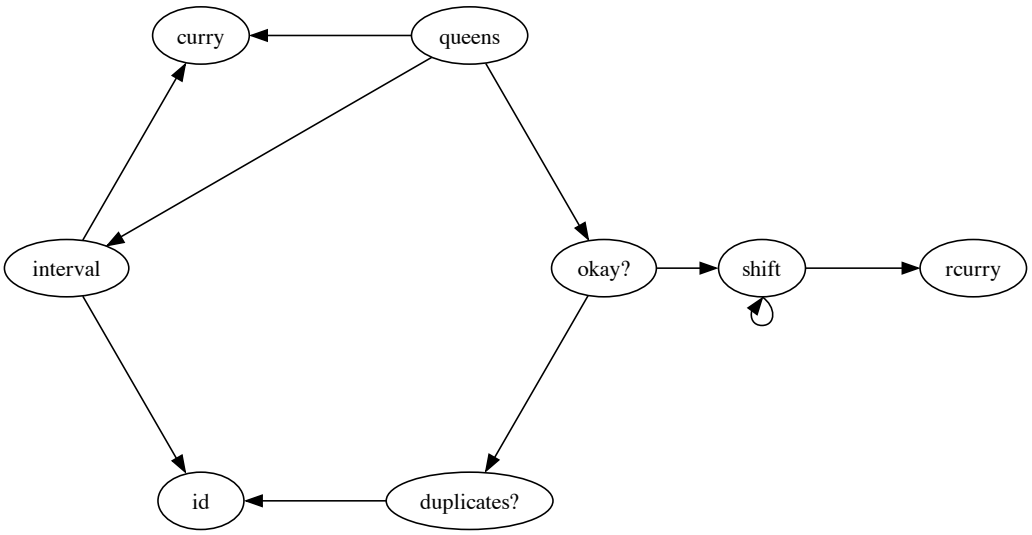


Figure 7: Call graph of the queens program

English was in fact Alcuin's native language but it probably wouldn't have helped you very much if had written this in English rather than Latin since he died a little over twelve centuries ago and English has changed quite a bit since then. Here's my own rough translation:

> A man needed to transport a wolf, a goat and head of cabbage across a river. The only boat he could find could not accommodate two of these. The task, then, was to transport all of them safely. Say, if you can, how they all managed to cross safely.

As with most applied problems, we are give a statement which is missing some important information, which we will have to fill in. We must, for example, give a precise meaning to the word "safe". Wolves eat goats. This practice is, from the point of view of the goat, unsafe.Goats eat cabbage. This practice is, from the point of view of the cabbage, unsafe. Of course we need to state all the negative assumptions as well.Of course we need to state all the negative assumptions as well. Wolves don't eat cabbages. Cabbages don't eat anything. Nothing eats wolves. Nothing eats itself. In Alcuin's time the vast majority of the population lived from agriculture. These facts must have been obvious to his readers. In our modern urban world it is better to make these assumptions explicit. There is an additional

assumption here, which, while not necessary to the formulation of the problem, is required for it to make sense. Wolves, goats and heads of cabbage are similar in size and shape, at least as far as fitting in boats is concerned. This makes me suspect that Alcuin wasn't really any more agriculturally inclined than I am. There is another implicit requirement in the problem, that the man and boat always travel together. We could add this also to the statement of the problem, but there is a better option. Do we really need both the man and the boat? Physically, of course we do. Mathematically, one of them is redundant, precisely because they always travel together. It doesn't matter much which we keep, but I will choose the boat.

We can now write a more precise form of the problem:

- We have a boat, cabbage, goat and wolf on one bank of a river.
- We want the boat, cabbage, goat and wolf on the other bank of the river.
- The allowed operations consist of transferring the boat and at most one of the other three from either bank to the other, without leaving the goat with either the wolf or the cabbage on opposite bank from the boat.

We can illustrate this with an undirected graph, in the sense described above, where the vertices represent the possible states of the system, i.e. who is on which bank, and the edges represent the allowed transitions. There are 2^4 possible ways to assign the boat, cabbage, goat and wolf to a river bank but six of those violate our safety constraint, leaving 10 vertices. Their labeling should be more or less self-explanatory. The symbol `‖` represents the river, and `b`, `c`, `g`, and `w` stand for "boat", "cabbage", "goat" and "wolf", respectively. There should be an edge connecting each pair of vertices where the boat and at most one of the cabbage, goat or wolf move from one bank to the other.

Once you have drawn the graph it's easy to find a solution to the problem. There are in fact exactly two solutions with the minimal possible number of crossings, which is seven. Of course with more realistic assumptions the set of solutions might be larger.

We will meet more practical applications of graphs later.

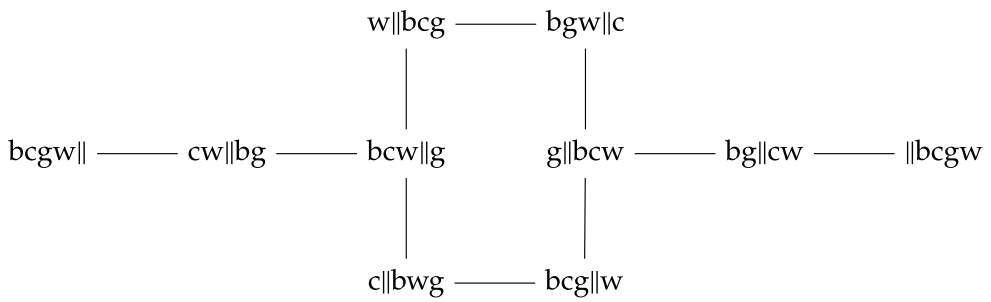


Figure 8: Transporting cabbages, goats and wolves

Interpretation

After this digression on trees and graphs we now return to our module enrollment problem.

If you've been reading very carefully you may have noticed that one of the ambiguities discussed previously has not been resolved, the one between inclusive and exclusive "or". Which one does $\vee$ indicate? The perspective taken by the theory of formal systems is that this distinction is not part of the language itself but rather of its interpretation. The language is described by its grammar and determines which statements are to be regarded as grammatically correct and how those statements are to be parsed but does not specify any particular interpretation of the language. The distinction between inclusive and exclusive "or" isn't needed for determining grammatical correctness or for parsing so it's not part of the language.

Note that this is different from the way we normally talk about natural languages. We regard the interpretation as part of the language for natural languages. The terms linguists use are syntax and semantics. Syntax determines grammatical correctness and parsing while semantics gives meaning to statements which are grammatically correct. A formal language is pure syntax.

People often refer dismissively to "arguments about semantics", which is odd since semantics is what gives meaning to statements.

In reality no one, except possibly as an example in a module like this one, would create a formal language without having an intended interpretation in mind though. One reason we make the distinction between language

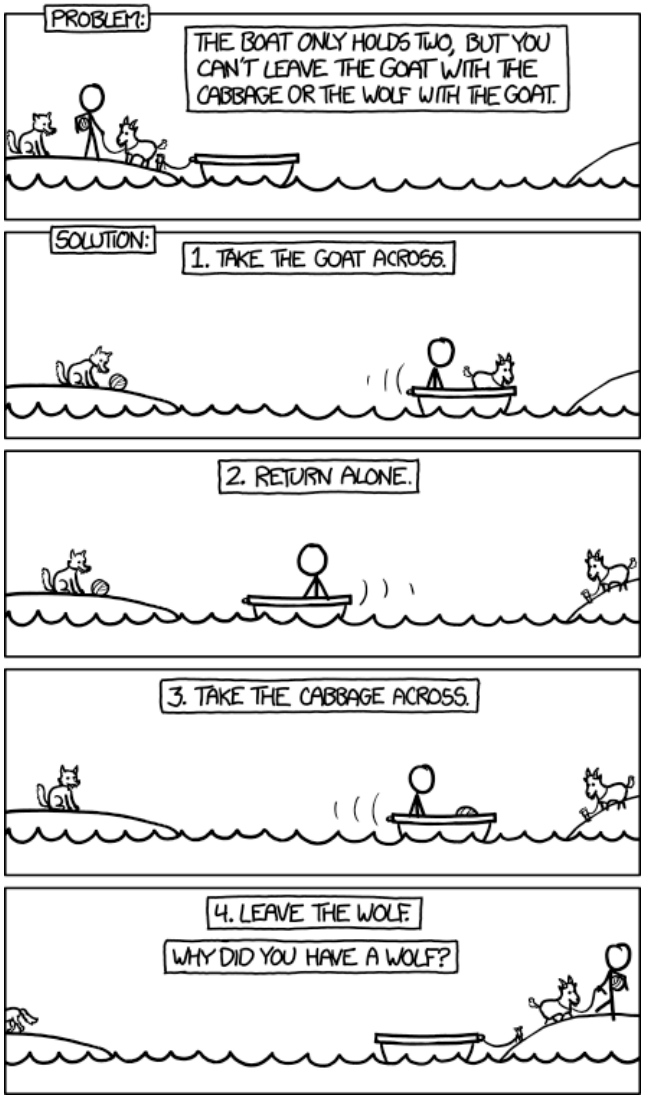


Figure 9: https://xkcd.com/1134

and interpretation is to allow the same language to have multiple interpretations.

The interpretation we'll give to our model module enrollment language is that $\vee$ is always an inclusive "or". The interpretations of $\wedge$ and $\neg$ are the usual meanings of "and" and "not", or at least what are usually referred to as their usual meanings. As mentioned earlier "and" can also be used in English in an exclusive sense, but we will always use it inclusively. With this interpretation the remaining ambiguity we saw earlier, concerning precedence between $\wedge$'s or between $\vee$'s is seen to be harmless, because $\wedge$ and $\vee$ are associative operators. We'll talk more about associativity when we discuss semigroups, monoids and groups later. Module names are interpreted as meaning that the student in question is taking that module.

If the "or" in "Probability and Statistics or Algebra and Geometry" in a course handbook was intended exclusively then in our formal language we will therefore need to replace it with something like "Probability $\wedge$ Statistics $\wedge$ $\neg$ Algebra $\wedge$ $\neg$ Geometry $\vee$ $\neg$ Probability $\wedge$ $\neg$ Statistics $\wedge$ Algebra $\wedge$ Geometry" in order to achieve the desired interpretation. If the "and"'s are also meant exclusively then we will need something even more complicated.

Strictly speaking our language has multiple interpretations, one for each student. We'll see more interesting examples later where it's useful for a language to admit multiple interpretations. We'll also see that this unavoidable for all but the simplest systems.

Expressiveness

It's possible for one language, with its intended interpretation, to be more expressive than another language, also with its intended interpretation, in the sense that any meaning which can be conveyed with the second can be conveyed by the first, but not vice versa. If we, for example, dropped the Boolean operator $\neg$ from our language above we would obtain a less expressive language because there would be some module selection rules we simply couldn't express.

On the other hand sometimes one language is larger than another without being more expressive. The language described above has parentheses, for

example, but would be equally expressive without them. We've already seen an example above of replacing a statement with parentheses with one without parentheses which has the same interpretation and this can in fact be done to any statement. Similarly our language doesn't have an exclusive "or" but we could add one, denoted for example by $\not\equiv$, without any gain in expressiveness. We've already seen an example of converting a statement with an exclusive "or" to one without any and this also can be done in general. A further possible addition to our language would be an "implies" operator. The usual notation for this operator is $\supset$. The statement "Statistics $\supset$ Probability" would mean that if a student is taking Statistics they are then also taking Probability, i.e. that Probability is a prerequisite or corequisite of Statistics. This also gives no gain in expressiveness. An equivalent statement without $\supset$ is "$\neg$ Statistics $\vee$ Probability". If this looks wrong then you may need to remind yourself of our precedence rules. Since $\neg$ is higher precedence than $\vee$ the statement will be parsed as "$(\neg$ Statistics$)$ $\vee$ Probability" rather than "$\neg$ (Statistics $\vee$ Probability)".

Is it worth adding language features which don't make a language more expressive? It often is, although such features are referred to dismissively as "syntactic sugar" by some authors. The equivalent versions of statements without the feature are often longer or harder to read than the versions with them, as we've seen. But there's a trade-off here. Adding language features may make it easier to craft a statement with your desired interpretation but it will make your language harder to parse and will also make it harder to reason about the language.

Parsing

I haven't given a purely formal description of our example module selection language. We'll see how to do that later. Hopefully I have described it in enough detail that you can recognise which statements are grammatically correct and which, like "Probability Statistics) $\wedge$ $\vee$ (Geometry $\neg$" are not grammatically correct, even though they are built from the same pieces. You can probably also mentally parse grammatically correct statements, at least if they're not too long and complicated. Whether you could write a parser for it is another matter. We have a certain level of parsing built in which is how even very small children can learn languages, but this process is mostly subconscious, which is why it's hard to write a parser even

for a language we would have no trouble parsing intuitively.

I'm not going to construct a parser for the module enrollment language described above. I could, but it would be quite complicated despite the apparent simplicity of the language. It would also be largely pointless, for reasons I'll explain soon. I will describe a parser for a closely related language though.

Infix, prefix and postfix

Our notation for the Boolean operators $\wedge$ and $\vee$ is what's called infix notation, where the operator is written between its operands. Alternatives are prefix notation, where it's written before the operands, or postfix notation, where it's written after them. The prefix version of "Probability $\wedge$ Statistics $\vee$ Algebra $\wedge$ Geometry" is

```
(∨ (∧ Probability Statistics) (∧ Algebra Geometry))
```

while the postfix version is

```
((Probability Statistics ∧) (Algebra Geometry ∧) ∨)
```

The prefix version may look familiar if you've ever seen any of the many variants of the programming language LISP, the second oldest programming language still in regular use.

The parentheses show the structure of the subphrases but aren't really necessary. There is no other way to split these statements. With prefix or postfix notation we also don't need precedence rules.

A parser for the prefix language

It's much easier to write a parser for a prefix or postfix language than an infix one. In fact here's a simple parser for the prefix version of our language, without the unnecessary parentheses.

The boring bit of the parser is the lexical analyser, the bit which separates the input stream into the three Boolean operators $\wedge$, $\vee$ and $\neg$ and the module names. We'll call these tokens. This is easy for our language, since we'll interpret any any string of characters other than $\wedge$, $\vee$ and $\neg$ as a module name. Checking whether the string corresponds to some actually existing

module is not the lexical analyser's problem, although we could have made it part of the lexical analyser's job. How much work, if any, the lexical analyser should do is a design decision. I will talk later about how a lexical analyser splits input into tokens but for now we'll just assume we have a lexical analyser and that our input is split into tokens, which the parser reads in one at a time.

Slightly simpler than an actual parser is a grammar checker. The only data structure this needs is a single integer, which we'll call the counter. The is initialised to 1. When the grammar checker reads an $\wedge$ or an $\vee$ it increments the counter. When it reads a module name it decrements the counter. When it reads a $\neg$ it does nothing. If the value of the counter reaches 0 at the end of the input, but not before, then the input is grammatically correct. Otherwise it isn't. Here's the input "$\vee$ $\wedge$ Probability Statistics $\wedge$ Algebra Geometry" together with the value of the counter at each point in the input:

```
1 ∨ 2 ∧ 3 Probability 2 Statistics 1 ∧ 2 Algebra 1 Geometry 0
```

In the theory of formal languages what I've just called a grammar checker is called a recogniser.

We can turn this into an abstract syntax tree by scanning through for sequences of tokens where the value of the counter remains at least as high as its value at the start of the sequence until the end of the sequence, where it's 1 lower. The seven sequences with this property in our example are

```
1 ∨ 2 ∧ 3 Probability 2 Statistics 1 ∧ 2 Algebra 1 Geometry 0
2 ∧ 3 Probability 2 Statistics 1
1 ∧ 2 Algebra 1 Geometry 0
3 Probability 2
2 Statistics 1
2 Algebra 1
1 Geometry 0
```

For each of them we have a node in our tree, which we will label with the first token of the sequence. Whenever one sequence contains in another we'll draw an arrow from the first to the second, unless there's an intermediate sequence, i.e. one which contains the second and is contained in the first. The result is the tree in the accompanying figure, which we saw earlier for the infix version of the same statement.

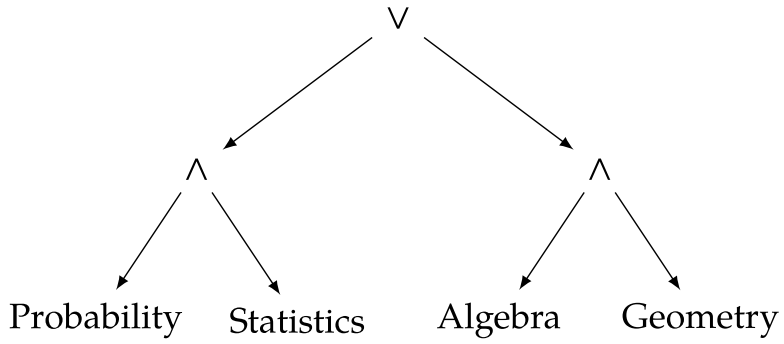


Figure 10: Syntax tree for ∨ ∧ Probability Statistics ∧ Algebra Geometry

The parse tree is similar to the one we saw earlier, but the order of the children is different, as shown in the accompanying diagram.

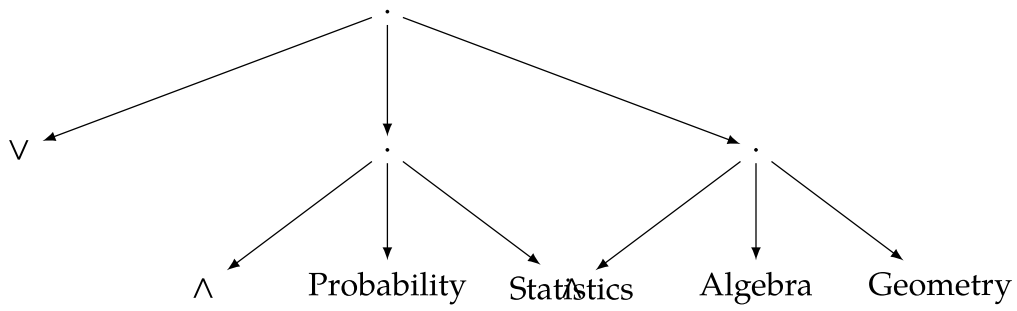


Figure 11: Parse tree for ∨ ∧ Probability Statistics ∧ Algebra Geometry

A parser for the postfix language

It's equally easy to write a parser for the postfix version. Again we'll start with a recogniser. The recogniser has a counter initialised to 0. When it reads a module name it increments the counter. When it reads an ∧ or an ∨ it decrements the counter. When it reads a ¬ it does nothing. If the counter remains positive until the end of the input, and is equal to 1 there, then the input is grammatically correct. Otherwise it is not.

Here is the input "Probability Statistics ∧ Algebra Geometry ∧ ∨" decorated with the values of the counter at each stage:

```
0 Probability 1 Statistics 2 ∧ 1 Algebra 2 Geometry 3 ∧ 2 ∨ 1
```

Constructing an abstract syntax tree is similar to the case of the prefix language. The nodes correspond to sequences of tokens where the value of the counter is 1 higher at the end than the start and is always higher in between than at the start, and we label each node with its final token. The abstract syntax tree for the input above is the same as for the corresponding input for the prefix parser, while the parse tree again differs only in the order in which the children of each node are listed.

Parser generators

It's probably not obvious that the parsers described above are correct. It's also probably not obvious how you would construct a parser for our original, infix, language. People realised early on that generating parsers is both a specialised skill and one which can be automated. There are programs, called parser generators, which take a description of a formal language and generate a parser for it. For them to be able to do this the description needs to be in a suitable format. In other words one needs a formal language for the description of formal languages. If you've written such a parser generator you can even apply it to its own language to generate another parser generator!

Writing a parser generator is generally harder than writing a parser, and proving a parser generator always generates correct parsers is generally harder than proving that any individual parser is correct, but the great advantage is that in principle you only need to do the work once.

A simple module selection checker

If we have a parser for our module enrollment language then we can write a recursive procedure for checking a student's module choices, starting from the abstract syntax tree. The procedure takes as input a node of the tree and has as output a Boolean, i.e. the value "true" or "false". When called on a node labelled by a module name it checks whether the student has selected the module, returning "true" if so and "false" if not. When called on a node labelled $\neg$ it calls itself on the node at the end of the outgoing arrow and returns "true" if that call returned "false" and vice versa. When called on

a node labelled $\wedge$ it calls itself on each of the nodes at the end of the two outgoing arrows and returns "true" if both of those calls returned "true" and "false" otherwise. When called on a node labelled $\vee$ it calls itself on each of the nodes at the end of the two outgoing arrows and returns "false" if both of those calls returned "false" and "true" otherwise.

Applying this procedure to the root of the abstract syntax tree for a module selection rule tells you whether the student's module selections are allowed by the rule.

This checker works equally well regardless of whether we chose the infix, prefix or postfix version of our input language, since they all have parsers which produce the same abstract syntax tree.

A simpler checker

Except for being recursive the procedure described above is fairly simple. It does depend on having a parser though, and parsers are not simple. For the postfix version of the language it's possible to avoid the parsing stage entirely and write a simple checker which works directly on the unparsed statements.

Our simple checker needs a stack, but no other data structures. A stack is a simple data structure with only three operations. We can push a value onto the top of the stack or pop the value currently at the top off of the stack. We can also check whether the stack is currently empty.

Our procedure starts with an empty stack and reads tokens one by one from the statement expressing the module rule. When it reads a module name it pushes a 0 onto the stack if the student has selected the module and pushes a 1 onto the stack if the student has not. When it reads a $\neg$ it pops a value from the top of the stack and pushes 1 minus that value onto the stack. When it reads an $\wedge$ it pops two values off of the stack and pushes their maximum onto the stack. When it reads an $\vee$ it pops two values off of the stack and pushes their minimum onto the stack. After reading all the tokens there is one value on the stack. The student's choices comply with the rule if and only if that value is 0.

Here is the statement "Probability Statistics $\wedge$ Algebra Geometry $\wedge$ $\vee$" decorated with the state of the stack after reading each token, if the student has

selected “Statistics” and “Algebra” but no other modules. To make things compact the stack is written horizontally, with the left hand side being the “top”.

```
Probability 1 Statistics 0 1 ∧ 1 Algebra 0 1 Geometry 1 0 1 ∧ 1 1 ∨ 1
```

The final value is 1, meaning the selection does not comply with the rule.

If you’ve been wondering what the counter in our recogniser for the postfix language represented you now have an answer: it’s the size of the stack. The contents of the stack depend on the individual student’s module choices but its size doesn’t.

One minor comment is that this module selection checker is using 0 and 1 as substitutes for the Boolean values “true” and “false”, in that order. With this convention “and” corresponds to a maximum and “or” to a minimum. Using 0 for “true” and 1 for “false” might seem odd but it has the advantage that $\wedge$ corresponds to a maximum and $\vee$ to a minimum, which is easy to remember because the arrow pointing upward means that we take the higher of the two numbers and the arrow pointing downwards means we take the lower one. Of course it would have been possible to use the reverse convention, with 0 for “false” and 1 for “true”, with relatively minor modifications to the algorithm.

You could write a similar checker for the prefix version of the language but it would have to read tokens in reverse order.

Idealised machines

It’s useful to think of various types of idealised machines, with varying levels of complexity, and classify computations by which of these idealised machines can perform them.

In this classification there’s a maximally powerful machine, which should be able to perform any calculation which can be performed. This is called a Turing machine. Those will be described much later in the module.

The simplest useful machine in this hierarchy is what’s called a finite state automaton. It has a single state variable, which can take only finitely many values, and must read its input one token at a time without backtracking. Our grammar checker for the postfix version of our language barely fails to

qualify. Its state is completely described by the counter but it can take any non-negative integer as its value and there are infinitely many non-negative integers.

A finite state automaton can be conveniently illustrated by a directed graph, where vertices correspond to possible states and edges correspond to the allowed state transitions. More information is needed to give a complete description of the finite state automaton, like which state is the initial state and which tokens in the input cause which state transitions. The accompanying figure gives an example.

We'll discuss such diagrams in more detail later but I'll give a quick explanation now. The alphabet accepted by this finite state automaton is the symbols P, S, A and G. Each state corresponds to a vertex, indicated by a circle or a double circle. The doubly circled vertices are accepting states, which means that if we are in one of those states when the input ends then the input is accepted. If the input ends when we're at a singly circled vertex then it is rejected. Each possible transition is indicated by an edge, i.e. an arrow. These edges are labelled by the symbols which cause the transition. The initial state is the one on the left with the arrow from nowhere. This automaton has been constructed in such a way that it will accept any input in which both P and S appear, or both A and G.

Intermediate in complexity between the finite state automaton and the Turing machine is what's called the pushdown automaton. This is an idealised machine whose only data structure is a single stack. Like the finite state automaton it must read its input one token at a time. The state of the machine is fully described by the contents of this stack. Our postfix module selection procedure is an example of a pushdown automaton.

There are a variety of visual representations of pushdown automata but none seem to be as standard as the one for finite state automata.

A hierarchy of languages

Corresponding to the hierarchy of idealised machine types there is a hierarchy of languages, with languages classified by which idealised machines are powerful enough to recognise the language, i.e. identify grammatically correct statements in the language. Languages which can be recognised

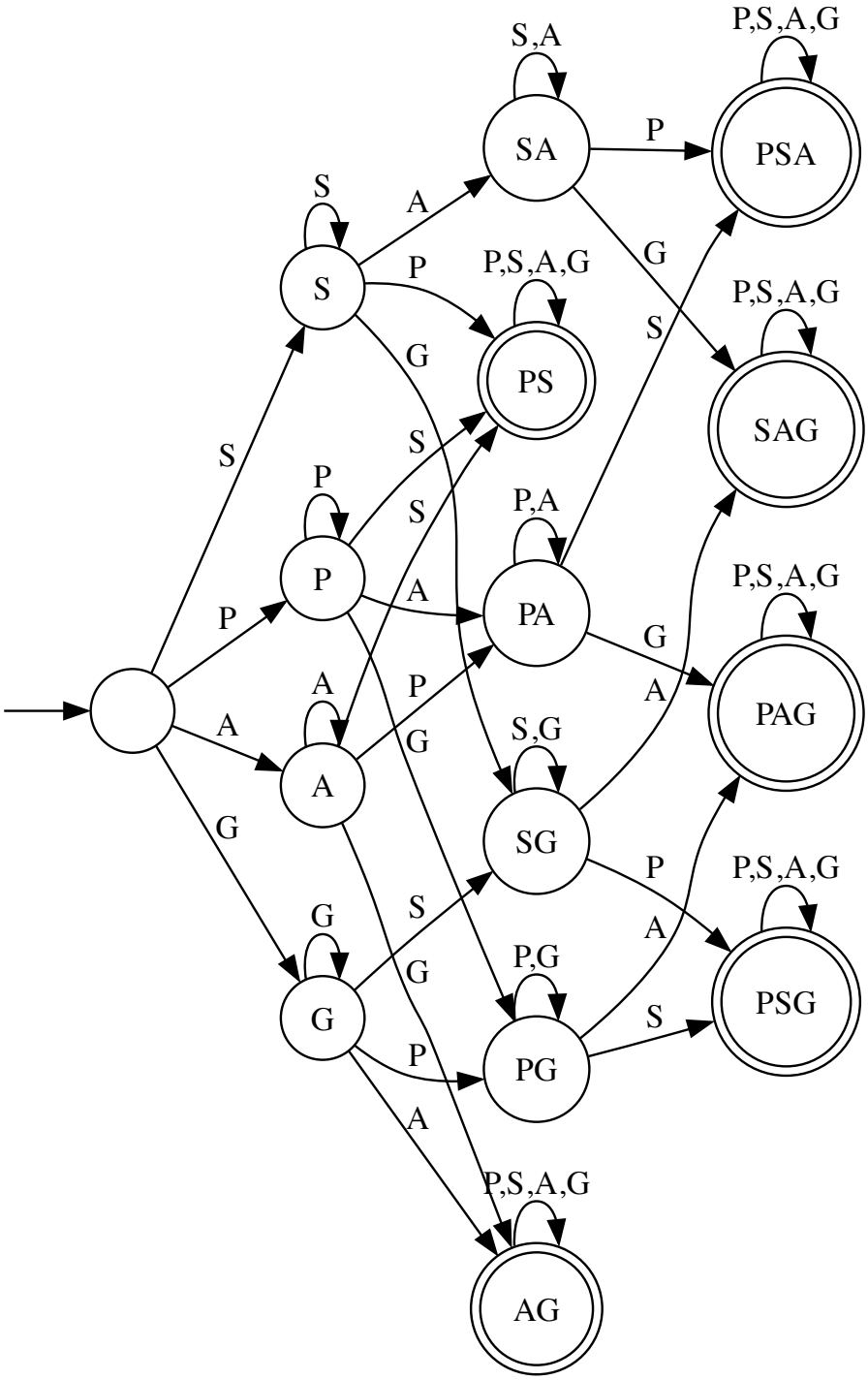


Figure 12: A finite state automaton

by a finite state automaton are called "regular". Languages which can be recognised by a pushdown automaton are called "context free". Languages which can be recognised by a Turing machine are called "recursively enumerable".

We know that the prefix and postfix versions of our module enrollment language are context free, because we've described an algorithm for recognising them which could be implemented by a pushdown automaton.

I didn't actually describe those algorithms in that way, instead using a non-negative integer as a state variable, but we can simulate non-negative integers with a stack. Zero is the empty stack. We increment by pushing something onto the stack and decrement by popping something off. What we push or pop is irrelevant. The size of the stack at each stage is what we earlier referred to as the counter.

We may strongly suspect that those languages are not regular, but we haven't proved that yet. Whether the infix version of our module selection language is context free is a question we'll return to later.

It's also possible to characterise these classes of languages purely in terms of their grammar, without reference to any idealised computing machine. This characterisation is useful because it's usually easier to write down a grammar for a language than to design an idealised machine to recognise it.

Satisfiability

One rather serious problem with our rule-based approach to the module enrollment problem is that it's possible inadvertently to create a set of rules which can't be satisfied by any set of modules a student might choose. A simple example would be the single rule "Probability $\wedge \neg$ Probability". Of course it's unlikely anyone would create such a rule accidentally but it's easy to imagine a set of rules each of which individually seems fine but which, taken together, have the unintended consequence of ruling out all possible module selections.

It's possible to prove that a set of rules can be satisfied by exhibiting a module selection which satisfies them. It's possible to prove that they can't be satisfied by checking all possible module selections. This works in theory

because the set of modules, and hence the set of sets of modules, is finite.

For any university with a realistic number of modules checking all possible module selections would never work from a practical point of view. Nevertheless we say the satisfiability problem in this context is decidable, because we could construct a Turing machine which would eventually answer the question. For more complicated languages the satisfiability problem is often undecidable even in theory.

Since our language is essentially that of zeroeth order logic we can borrow satisfiability checking algorithms from there. These methods are faster in practice than checking all possibilities but their theoretical worst case complexity is poorly understood.

I've just described satisfiability as a property of a statement in a language, but this isn't quite correct. It's a property of the statement, language and interpretation. Without the interpretation we wouldn't be able to determine when the statement is true.

Tautologies and consequences

A less serious problem is that it's possible to specify redundant rules. The most extreme form is a rule which is always satisfied, like "Probability $\vee$ $\neg$ Probability". These are called tautologies. The problem of identifying tautologies is in some sense dual to that of identifying unsatisfiability. Instead of looking for rules which can never be satisfied we're looking for ones which are always satisfied.

A tautology is a special case of a consequence. One statement is a consequence of others if it is always satisfied whenever they are. A tautology is a statement which is a consequence of the empty set of statements. The question of whether a statement in our language is a tautology is decidable, at least in a theoretical sense. More generally, the question of whether one statement is a consequence of a list of other statements is decidable, in the same sense. Whether these questions are decidable in a practical sense is another matter entirely.

Rules of inference

The definition for a consequence given above requires checking a very large number of possibilities. To verify that "Probability ∧ Statistics ∨ Algebra ∧ Geometry" is a consequence of "Probability ∧ Statistics" we would have to check all possible module selections and confirm that all the ones which satisfy the second statement also satisfy the first one. That's tedious and unnecessary. If A and B are grammatically correct statements then "A ∨ B" is always a grammatically correct statement and is a consequence of A and also a consequence of B. Transformations like this which take statements and give you consequences are called "rules of inference". The soundness, or validity, of a rule of inference, the property that the statements they produce are actually consequences, depends on the interpretation. The rule for ∨ given above is a sound rule of inference for our system with its intended interpretation.

Writing down sound rules of inference can be tricky. It might seem obvious that if A and B are each grammatically correct statements then "A ∧ B" is a grammatically correct statement and that A and B are both consequences of it. This unfortunately isn't true. "Probability ∧ Statistics ∨ Algebra ∧ Geometry" is grammatically correct statement, as are "Probability" and "Statistics ∨ Algebra ∧ Geometry". The second of these is indeed a consequence of "Probability ∧ Statistics ∨ Algebra ∧ Geometry" but the first is not. There are module selections for which the statement "Probability ∧ Statistics ∨ Algebra ∧ Geometry" is satisfied but the statement "Probability" is not. The student could, for example, select Algebra and Geometry, and possibly various other modules, but not Probability. The problem here is that this ∧ is an unnatural place to break the expression "Probability ∧ Statistics ∨ Algebra ∧ Geometry". It's possible to express this in terms of the abstract syntax tree. Breaking a statement into two pieces using an ∧ at the root of its abstract syntax tree is safe. Breaking it at an ∧ elsewhere in the tree is dangerous.

Changes to the language can help. For the prefix version of the grammar it is true that if A and B are grammatically correct then "∧ A B" is grammatically correct and they are consequences of it. The same is true for the postfix version, except now the consequence is "A B ∧". For the fully parenthesised infix language it's true that if A and B are grammatically correct then so is "(A ∧ B)" and they are consequences of it. In none of these

cases does the rule of inference need to refer to the abstract syntax tree. Our choice of language, with infix notation and with parentheses used only where needed to override precedence rules, turns out to be a particularly unfortunate one.

Formal systems

A formal system is a language defined by a grammar together with a set of axioms, and a set of rules of inference. The rules of inference should refer only to the language and grammar, not any particular interpretation. An interpretation is sound if the axioms are true in that interpretation and the rules of inference when applied to true statements generate only true statements. Statements which can be derived from the axioms using the rules of inference are called theorems and any such derivation is called a proof of the theorem. Theorems are true in any sound interpretation. A true statement in a particular sound interpretation need not be a theorem though. This will certainly be the case if there is another sound interpretation in which the statement is false.

The above definition of theorem and proof are the one used by logicians. Mathematicians tend to use the terms somewhat differently. Mathematicians typically refer to something as a theorem only after a proof has been found. They refer to a proof in the logician's sense as a formal proof. By an informal proof they mean a convincing argument that the statement is true in the intended interpretation or interpretations. This is necessarily somewhat vague. What's convincing to one person may not be to another. More worryingly, there's no way to compare interpretations directly. The writer and reader of an informal proof may have subtly different interpretations and the statement may be true in the writer's interpretation and false in the reader's. Intermediate between formal and informal proofs we have semiformal proofs. A semiformal proof is a convincing argument that a formal proof exists. This might include, for example, an algorithm for producing such a formal proof. That's a viable strategy in cases where it's easier to verify that the algorithm is correct than actually to run it. We'll see examples later.

Should you have more faith in a formal proof than an informal one? Possibly, but not necessarily. Formal proofs have many advantages. They can be

checked mechanically. They imply that the statement is true in any sound interpretation. But mechanically checking only works if the checking algorithm is correct. The interpretation is only sound if the axioms are true and the rules of inference preserve truth. What assurance do we have on any of these points? Usually an informal proof! Formal proofs therefore don't really rest on any firmer philosophical foundations than informal ones. They can still be practically useful though. Checking the soundness of an interpretation or the correctness of a verification algorithm is generally a lot of work but it only needs to be done once. In this way the situation is analogous to the one we encountered earlier with parser generators.

In reality we typically start with a language and interpretation and then look for a set of axioms and rules of inference. We shouldn't include any false axioms or rules of inference which allow us to derive false statements from true ones. Otherwise we wouldn't have a sound interpretation. It would be nice to have finite sets of axioms and rules of inference but sometimes it's convenient to consider systems where one or both of those sets are infinite. We should at least insist on an algorithm for deciding whether or not a statement is an axiom or can be derived from a list of other statements via the rules of inference though.

Ideally we'd like a set of axioms and rules of inference which are large enough so that all true statements are theorems. For our module enrollment language it's possible to accomplish this but there are many settings where it's not possible. In fact it's not even possible in what's just about the simplest mathematical setting imaginable: the arithmetic of non-negative integers!

Sets

I've referred to sets informally several times above. All of the sets involved were finite, which is why all the questions we considered were decidable, again in a theoretical sense. There are infinite sets lurking in the background though. The set of all possible statements in our language is infinite. It is in some sense only mildly infinite though. More specifically, it is countable, a term we'll define later. We actually considered multiple different languages built from the same set of tokens. The infix, prefix and postfix languages are distinct languages. How many languages are there?

This requires a definition of language, which we haven't given yet, but there are infinitely many, and even uncountably many, even if we restrict to those based on the same finite set of tokens. There are however only countably many grammars so there are languages which cannot be described by a grammar. There are also only countably many Turing machines so there are languages which can't be recognised by any Turing machine, i.e. are not recursively enumerable.

Later we'll see a formal language to describe the theory of sets. As we've just seen though, it can't describe each individual set, because there will only be countably many statements and the number of sets can't be countable. Set theory is nice and intuitive as long as we restrict ourselves to finite sets but rapidly becomes weird when we have to consider infinite sets.

A regular language

The module enrollment problem we've been discussing requires input from staff, about which combinations of modules students should be able to take, and from students, about which modules each student wants to take. So far we only have a language for the input from staff. In reality the students would probably select modules from some sort of web interface, but for the implementer it would be much easier just to provide a language for their input as well. The simplest such language would have statements which are just lists of modules. The statement "Statistics Algebra", for example, would have the interpretation "I want to take Statistics and Algebra and nothing else".

If our language includes all such lists of modules then no parsing is really needed. The lexical analyser, which splits the input into tokens, i.e. module names, does all the work.

There's another option though. At the point where students are entering their module selections the staff have already entered all the information about allowed combinations. We could define a language consisting of precisely those module lists which are allowed. The information collected from the staff implicitly gives this language a grammar and grammatically correct just means allowed by the module selection rules. Of course any change to those rules gives us a new language.

What sort of language is this? It turns out to be regular. It is possible to create a finite state automaton which recognises it. In fact the example of a finite state automaton I gave you earlier is essentially the one which enforces the rule "Probability $\wedge$ Statistics $\vee$ Algebra $\wedge$ Geometry". I just shortened each of the module names to just their initial letter to avoid clutter in the diagram.

One way to construct a module enrollment system would be to use the following components:

- A parser generator. There are parser generators freely available which efficiently generate efficient parsers so we don't need to write anything.
- A simple lexical analyser. It just needs to distinguish module names from the Boolean operators $\wedge$, $\vee$ and $\neg$ so it's easy to write. It's helpful if it has an option to throw an error whenever it sees a Boolean operator. That way it can be used, with the option unset, for input from staff entering module selection rules and, with the option set, for input from students selecting modules.
- A grammar for the module rule language, which is the same as the language of the propositional calculus, written in the language which the parser generator accepts as input. This is very easy to write since the grammar is very simple.
- The parser generated from this grammar. This may be complicated, but it's generated for you by the parser generator.
- A procedure which converts parsed statements to a grammar for the language of module selections for which those statements are true. This is the hardest part and unfortunately is very hard to do efficiently. It's possible to arrange that the grammar is a regular grammar.
- The parser generated by the parser generator from that grammar. Again, this parser will probably be complicated but it's generated for us by the parser generator. Since the grammar is regular it could generate a finite state machine parser, but might choose not to. The actual parsing isn't really what's needed, just the check that the module selection is grammatically correct.

I'm not saying you should construct a module enrollment system this way,

merely noting that you can.

Conclusion

This introduction was intended mainly to introduce a cast of characters which will play a more prominent role later in the module. Of particular importance are formal languages, algorithms and computability, zeroeth and first order logic, grammars, quantifiers, variables, parsing, trees and graphs, interpretations, extensions of languages and expressiveness, the hierarchies of languages and idealised machines, satisfiability, tautologies and consequences, integers and sets.

One important thing to take away from this is that formal languages do not emerge fully formed from a vacuum. They are designed by humans. They may be intended to be written and read by humans, by computers, or by both. That design process involves a number of compromises, for example between making it possible to express simple ideas with similarly simple statements on the one hand and making statements easy to parse on the other. Formal languages tend to be annoying to work with. Understanding those design trade-offs doesn't necessarily make them less annoying, but it may at least make the reasons for those annoying aspects clearer.

Languages

We'll discuss formal languages in more detail later, after talking about logic and set theory, but we need at least a rudimentary understanding of formal languages in order to formalise logic and set theory.

A grammar example

`bc` is an arbitrary precision calculator. It's part of the POSIX specification for Unix operating systems. That specification not only requires a `bc` program to be present but also gives a minimal grammar which it must recognise, which makes it useful as an example. Even this grammar is a bit too complex as an initial example though so I'll only use pieces of it.

Grammars are usually specified in some variant of Backus-Naur form. The specification uses the variant used by yacc, a parser generator. Internet

RFC's, the documents which govern the internet, use a different variant, or really several different variants. I'm going to use yet a different variant, derived from one of the parser generators used by the Racket programming language, which is more concise than the others mentioned above. All of these are similar enough that if you have experience with one you can usually figure out any of the others without even needing to consult the documentation, unless the grammar writer has used some really obscure features of a particular one.

Terminology

We need a bit of terminology in order to talk about this and other formal languages.

Languages have an alphabet consisting of tokens. These might be single characters or might be strings. Converting a list of characters into a list of tokens is the job of the lexical analyser. The tokens belong to sets, called terminal symbols. There are also nonterminal symbols, which we'll get to later.

In the example above the alphabet consists of 17 tokens, each a single character, the decimal point and 16 digits, and there's a terminal symbol for each of these.

In this case the lexical analyser doesn't really have anything to do, but we could have divided the work between the lexical analyser and the parser differently. For example, we could have put all the digits into a single token, `DIGIT`, in which case the grammar would have looked like

```
number : integer | "." integer | integer "." | integer "." integer
integer : DIGIT | integer DIGIT
```

The standard notational convention is that names of terminal symbols are written in upper case. The parser now doesn't need to concern itself with which characters are digits as that's handled by the lexical analyser.

We could shift even more work from the parser to the lexical analyser, by deciding that uninterrupted strings of digits are tokens, with the symbol `INTEGER`. The parsers grammar is then just a single line.

```
number : INTEGER | "." INTEGER | INTEGER "." | INTEGER "." INTEGER
```

In this case, unlike the previous ones, there are infinitely many tokens. This is allowed, but we'll still insist on having only finitely many symbols. When there are only finitely many tokens it's clear that we can figure out which symbol a token belongs to just by comparing it to the finite lists for each symbol. When we have finitely many symbols but infinitely many tokens there must be at least one symbol with finitely many tokens, so we can't just examine the lists. There needs to be an algorithm for recognising tokens. We'll discuss what kinds of algorithms are allowed later in the chapter on regular languages.

The symbols we've just discussed, the ones which are sets of tokens are called terminal symbols or just terminals. There are other symbols, conveniently called nonterminal symbols or just nonterminals. In the original grammar above `number`, `integer`, and `digit` were nonterminals. In fact the nonterminals are always precisely the things you see listed on the left hand sides of all the grammar rules.

One of these symbols has a special status. It is called the start symbol. In the example above `number` is the the start symbol. The notational convention I'm using is that the start symbol is always the one on the left hand side of the first rule in the list.

A context free grammar is a finite set of grammar rules, also sometimes called production rules. Each of these grammar rules describes possible ways to build up a nonterminal symbol from other symbols, which might be terminal or nonterminal. For example, an `integer` is either just a `digit` or is an `integer` followed by a `digit`. The `|` character separates the distinct possibilities in each case.

You can see from the rule for `integer` that when I wrote that "each of these grammar rules describes possible ways to build up a nonterminal symbol from other symbols" I didn't mean the word "other" to exclude the possibility of the same symbol occurring on both the left hand side and the right hand side of a rule. In other words, rules can be recursive. Indeed almost all interesting languages have recursive rules.

Example, continued

You shouldn't assume just because a name is familiar that it means what you think. In our example

```
number : integer | "." integer | integer "." | integer "." integer
integer : digit | integer digit
digit   : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
        | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F"
```

`ACAB` is a `number` while `-7` and `5,011,400` are not. There are reasons for this. `A` through `F` are classed as digits to allow for hexadecimal representations of numbers. Disallowing the commas which traditionally separate groups of three digits is a design decision. It simplifies processing and avoids the awkward fact that most of the non-English speaking world uses dots instead of commas, while India uses commas but places them differently.

The minus sign isn't needed because `bc` is a calculator and part of its grammar that I omitted earlier is

```
expression : number | "(" expression ")" | "-" expression
          | expression "+" expression | expression "-" expression
        | expression MUL_OP expression | expression "^" expression
```

This is another recursive rule. `MUL_OP` is in upper case so we recognise that it must be a terminal symbol. As you might guess from the name it includes the token `*` for the multiplication operator but it also includes a couple of tokens. From this rule we see that a `number` is an `expression` and `-` followed by any `expression` is an `expression` so `-7` isn't a `number` but it is an `expression`. You might have noticed that a `-` appears in two different possible expansions of `expression`. In addition to the expansion `"-" expression` there's also `expression "-" expression`, which would allow, for example, `27-9`.

In any case, lets try the generative grammar approach and generate some `NUMBER`s. We'll start from the rule for `NUMBER` and pick possibilities at random each time we have to expand a nonterminal or choose a token for a terminal. Each line will be the result of doing this to the previous line.

```
number
integer
```

```
digit
7
```

So `7` is a `number`. Let's try again.

```
number
. integer
. digit
. B
```

So `.B` is also a `number`. Another two attempts:

```
number
integer
digit
5
```

```
number
integer . integer
digit . integer
digit . integer digit
digit . digit digit
5 . digit digit
5 . C digit
5 . C C
```

So `.B`, `5` and `5.CC` are `numbers`. Note that the spaces between symbols above, and in the specification are just there to improve readability and are not part of the string we're generating.

Natural languages

Generative grammar was originally developed for natural languages rather than for formal languages. As it turns out, natural languages are rather messy and this model doesn't fit them fully, but you've been exposed to them for larger so it may be easier to understand some of the concepts in that context. The usual way to do this is to take words as tokens. What people in formal languages call an "alphabet" is then not what we would normally think of as the alphabet of the language but is in fact its full vocabulary.

The lexical analyser splits a stream of characters into words. This is easy in a language like English, where we put spaces between words, but more difficult in a languages like Japanese, which does not. Even for English things are a bit more complex than just splitting at white space, but let's not worry about this now. The set of English words is theoretically finite, but not in any particularly useful way. There are languages, e.g. Turkish, where the set of words is arguably infinite. Terminal symbols represent types of words, often called "parts of speech", like nouns, verbs, adjectives, etc. There is a finite, and indeed rather small, set of such terminals. In formal language theory we generally assume that each token belongs to one and only one terminal. Some natural languages more or less satisfy this condition. English very much does not. The word "fast", for example, can be a noun, verb, adjective or adverb. Having raised these issues I will now ignore them, and pretend we have a lexical analyser which can reliably split our input into tokens, i.e. words, and uniquely identify their terminals, i.e. parts of speech.

A full grammar for a language like English would be very complicated but it's surprisingly easy to write down simplified grammars which are nonetheless sufficient to parse even fairly complex sentences. The following example is from Masaru Tomita:

```
sentence : noun_phrase verb_phrase | sentence prep_phrase
noun_phrase : NOUN | DETERMINER NOUN | noun_phrase prep_phrase
prep_phrase : PREPOSITION noun_phrase
verb_phrase : VERB noun_phrase
```

There are four terminals, `NOUN`, `DETERMINER`, `PREPOSITION` and `VERB` and four nonterminals, `sentence`, `noun_phrase`, `prep_phrase`, and `verb_phrase`. This grammar is clearly recursive, since `sentence` is defined in terms of itself, as is `noun phrase`. Grammars for natural languages are always recursive.

There are obviously important parts of English, like pronouns, adjectives and adverbs, which are missing, but we can still parse sentences like "Masaru saw a man in the apartment with a telescope" with only this fragment of English grammar. One possible parse tree is given in the figure. There are many more parse trees for this sentence though. You might want to see how many you can find.

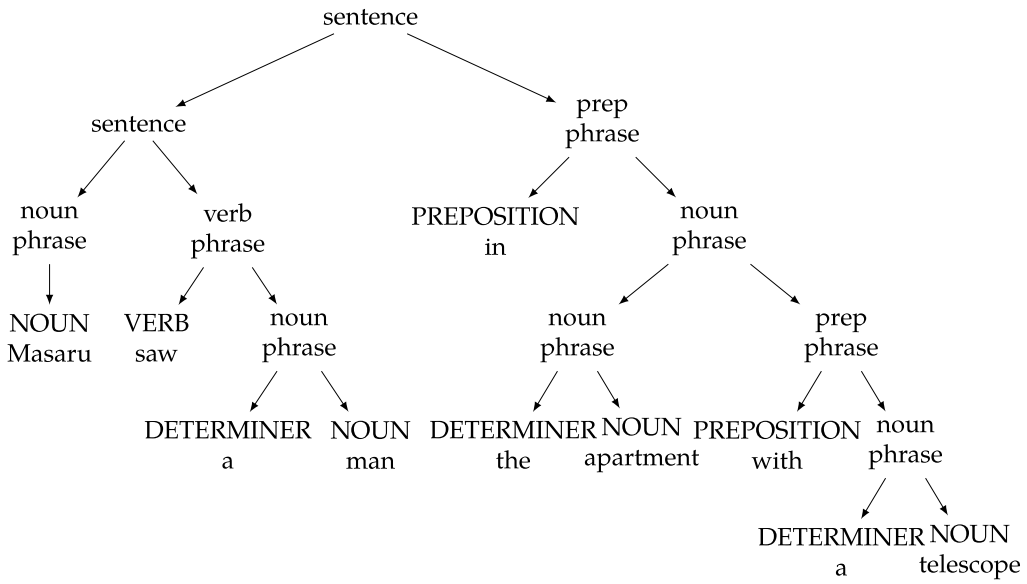


Figure 13: Full parse tree for Tomita's example

A formal language for linear equations

As another example, consider a language for systems of linear equations, with the grammar

```
equations : equation | equations "," equation
equation : side "=" side
side : term | side operator term
term : integer | variable | integer variable
operator : "+" | "-"
variable : "w" | "x" | "y" | "z"
integer : "0" | positive_integer | "-" positive_integer
positive_integer : positive_digit | positive_integer digit
digit : "0" | positive_digit
positive_digit : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

As usual, I'll assume that the lexical analyser strips out whitespace.

The definition of integers is borrowed from an earlier example. I've only allowed four variables. For real applications we would probably want more. An example of a string in this language would be $3z - 1 = x, 2y = 1$. Its

parse tree is as in the figure.

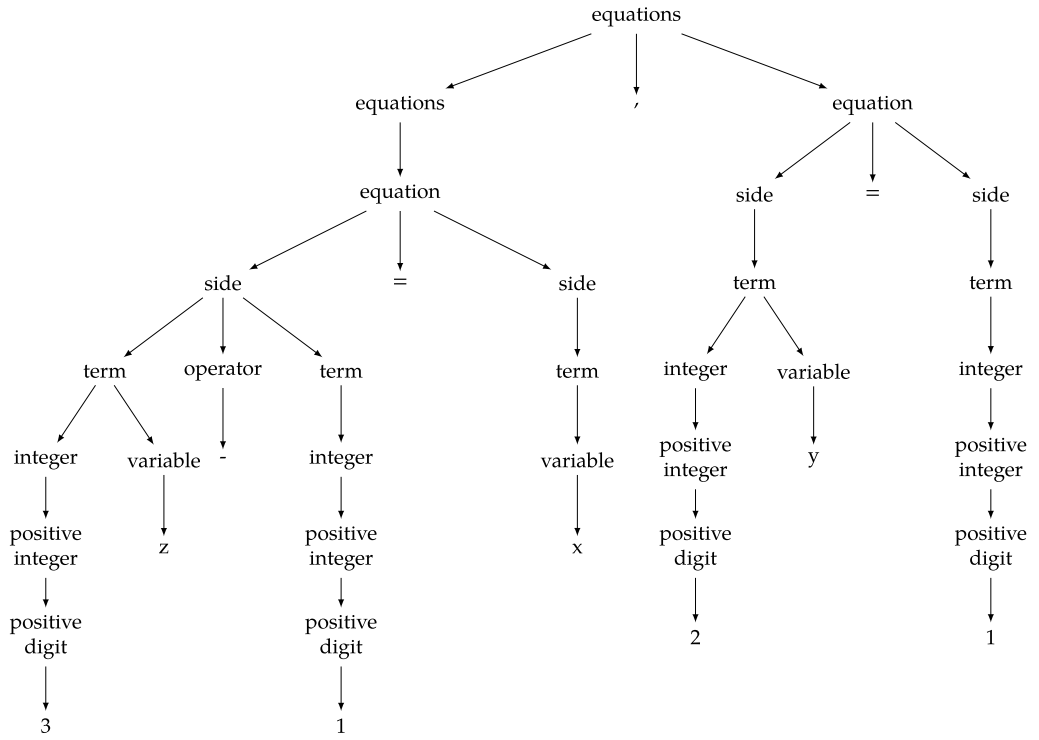


Figure 14: Full parse tree for 3 z - 1 = x , 2 y = 1

Thinking backwards

When I discussed languages earlier it was from the point of view of parsing or at least recognising them. We receive a list of tokens from the lexical analyser and want to piece them together into larger and larger phrases until we have one phrase encompassing the whole input. This process should be entirely deterministic and should terminate at some point.

The way the grammar describes the language is completely the opposite. Its starting point is the start symbol. It then "expands" that into a list of other symbols, which are then further expanded. We can only expand non-terminals. Once we reach a terminal we have to choose from among tokens composing that terminal and no further expansion is possible. I wrote the word "expand" in quotation marks because the "expansion" might not be

any larger than what we started with–it could be a single symbol–and it could even be smaller–an empty list of symbols.

You should think of the grammar as describing a method for generating elements of our language. A list of tokens belongs to the language if and only if this process of expansion starting from the start system could eventually produce it. Interesting languages tend to be infinite and the expansion process described above is nondeterministic because of the multiple possibilities for expanding each symbol, and it need not terminate, so this isn't a definition which is testable in any obvious way even when you have the full grammar specification.

As an approach to linguistics this is called "generative grammar". It was developed by Dakṣiputra Pāṇini about two and a half millenia ago.

The language of balanced parentheses

Another interesting example language is the language of balanced parentheses, also known as the von Dyck language. This language has only two tokens, "(" and ")". It consists of those lists of tokens where we can match open and close parentheses in such a way that they are nested and occur in the correct order.

Every element of this language has the same number of ('s and)'s, but that's not sufficient. ")(", for example, does not belong to the language. We could also allow pairs of "["'s and "]"'s or "{"'s and "}"'s, in addition to ('s and)'s, but won't, or at least not yet.

A grammar for the language of balanced parentheses is

```
a : | b ;
b : "(" ")" | "(" b ")" | "(" ")" b | "(" b ")" b
```

Note the empty space between the `:` and `|` in the rule for `a`. This indicates that an empty list of symbols is an acceptable expansion for `a`. We need this because an empty string has balanced parentheses, trivially, according to our definition.

Every list generated by these rules has balanced parentheses. Less obviously, every element of the language can be generated by these rules. Still less obviously, it can be generated in only one way. I won't give the proof

but here's the key idea: Every non-empty element of the language starts with a `(`. This `(` must have a matching `)`. The list of tokens in between has matching parentheses, as does the list after. Either or both of those lists could be empty.

The language of palindromes

Palindromes are words or sentences which are the unchanged by reversing the letters. Usually we ignore capitalisation, spaces and punctuation, so "Anna", "Bob", "Eve", "Hannah" and "Otto" are palindromes, as are "Cigar? Toss it in a can. It is so tragic." and "NIΨON ANOMHMATA MH MONAN OΨIN".

A grammar for palindromes is

```
palindrome : | non_empty_p
non_empty_p : "a" non_empty_p "a" | "b" non_empty_p "b"
            | ... | "z" non_empty_p "z" | "a" "a"
            | "b" "b" | ... | "z" "z" | "a" | "b" | ... | "z"
```

This assumes that the lexical analyser strips out everything which isn't a letter of the English alphabet and changes every upper case letter to the corresponding lower case letter. The "..." isn't really part of our language for specifying grammars. I just used it to prevent the specification from being too long.

It's important to realise that the following would not be a grammar for the language of palindromes.

```
palindrome : | non_empty_p
non_empty_p : letter non_empty_p letter | letter letter | letter
letter : "a" | "b" | ... | "z"
```

The problem with this grammar is that in the expansions `letter non_empty_p letter` and `letter letter` the two different occurrences of `letter` could expand to different letters, so we wouldn't have a palindrome.

More numerical examples

If you only want to allow decimal integers and you want to allow them to be negative you could use the following grammar:

```
integer : "0" | pos_integer | "-" pos_integer
pos_integer : pos_digit | pos_integer digit
digit : "0" | pos_digit
pos_digit : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

We've eliminated the digits needed for hexadecimal and allowed our `integers` to be negative. There are some further changes. `007` is a perfectly good `integer` according to `bc`'s grammar but is now disallowed. The only `integer` allowed to begin with a `0` is now `0` itself. We don't allow `-0` either. These are design decisions, made to ensure that in this grammar there is one any only one way to represent each integer as an `integer`.

It's possible to encode some basic arithmetic in a grammar. Consider, for example, the following grammar.

```
even_integer : "0" | pos_even_integer | "-" pos_even_integer
pos_even_integer : pos_even_digit | pos_integer even_digit
pos_integer : pos_digit | pos_integer digit
even_digit : 0 | pos_even_digit
pos_digit : pos_even_digit | pos_odd_digit
pos_even_digit : "2" | "4" | "6" | "8"
odd_digit : "1" | "3" | "5" | "7" | "9"
```

The `even_integers` described by this grammar are precisely the even integers. This relies on the fact that an integer is even if and only if its last digit is even.

You can check divisibility by three as well.

```
multiple_of_3 : "0" | pos_integer_0_mod_3
              | "-" pos_integer_0_mod_3
pos_integer_0_mod_3 : pos_digit_0_mod_3
                    | pos_integer_0_mod_3 digit_0_mod_3
                    | pos_integer_1_mod_3 digit_2_mod_3
                    | pos_integer_2_mod_3 digit_1_mod_3
pos_integer_1_mod_3 : digit_1_mod_3
```

```
                    | pos_integer_0_mod_3 digit_1_mod_3
                    | pos_integer_1_mod_3 digit_0_mod_3
                    | pos_integer_2_mod_3 digit_2_mod_3
pos_integer_2_mod_3 : digit_1_mod_3
                    | pos_integer_0_mod_3 digit_2_mod_3
                    | pos_integer_1_mod_3 digit_1_mod_3
                    | pos_integer_2_mod_3 digit_0_mod_3
digit_0_mod_3 : "0" | pod_digit_0_mod_3
pos_digit_0_mod_3 : | "3" | "6" | "9"
digit_1_mod_3 : "1" | "4" | "7"
digit_2_mod_3 : "2" | "5" | "8"
```

The `multiple_of_3`s are just the multiples of three.

How far can we go in this direction? Can we express divisibility by any integer purely in grammatical terms? As it turns out, yes. Since we can express divisibility can we write down a grammar for prime numbers? In this case the answer is more complicated. We can't construct such a grammar using only rules of the type considered above but we can if we allow more complicated rules, which replace a list of symbols with another list of symbols rather than just replacing a single symbol with a list of symbols.

Ambiguous grammars

Almost all of the grammars we've considered so far have been unambiguous, in the sense that there's only one abstract syntax tree we can get from any given input. One exception was the fragmentary grammar for English. Grammars for natural languages are almost always ambiguous. The other exception was the language for `expressions` which we extracted from the grammar of the `bc` utility.

```
expression
expression + expression
number + expression
integer + expression
digit + expression
1 + expression
1 + expression + expression
1 + number + expression
```

```
1 + digit + expression
1 + 2 + expression
1 + 2 + number
1 + 2 + digit
1 + 2 + 3
```

and another is

```
expression
expression + expression
expression + number
expression + digit
expression + 3
expression + expression + 3
number + expression + 3
digit + expression + 3
1 + expression + 3
1 + number + 3
1 + digit + 3
1 + 2 + 3
```

These differ not just in the order in which we expanded symbols but in how the expression `1 + 2 + 3` is broken up into phrases. In the first one `1` and `2 + 3` are expressions joined by a `+`. In the second `1 + 2` and `3` are expressions joined by a `+`.

You can see that these are in fact distinct parsings by looking at the corresponding parse trees.

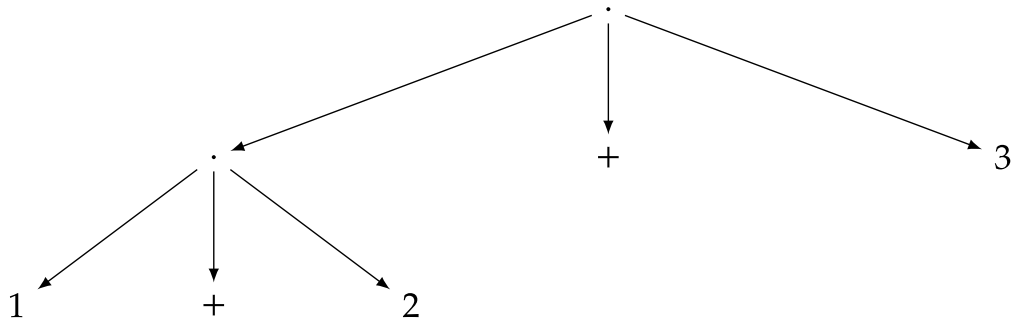


Figure 15: First parse tree for 1 + 2 + 3

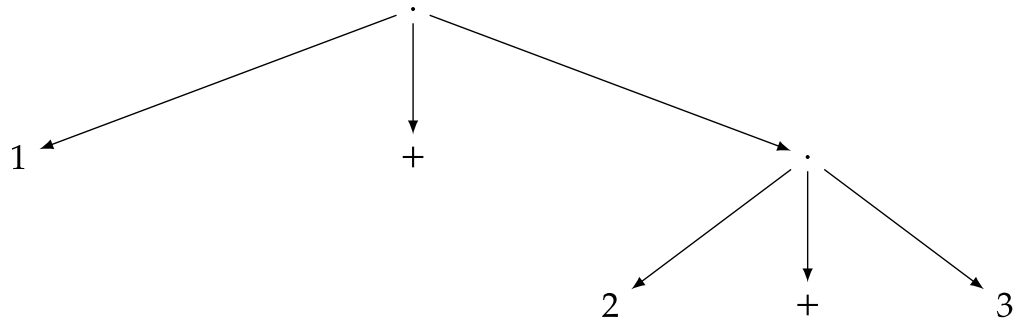


Figure 16: Second parse tree for 1 + 2 + 3

Strictly speaking these aren't full parse trees, since they don't show all the symbols which are expanded when generating the input. The full version, with nodes labelled by symbols, is given just for the first of these.

Ambiguous grammars are allowed. In fact there are context free languages for which no unambiguous grammar exists. It's also possible, and indeed common, for an ambiguous grammar and an unambiguous grammar to define the same language. It's often easier to write an ambiguous grammar for a language and often easier to analyse an unambiguous one. In fact the specification for `bc` doesn't require the particular grammar given in the specification, merely that whatever grammar is used should recognise the same language as this one generates. There are unambiguous grammars for this language and an implementation which used one would still be compliant.

Constructing a parser from a grammar

Can we construct a parser from a grammar description of the type we've just described? Yes. We can even do so in a way which is reasonably efficient. Unfortunately that way is also very complicated to describe. If we're willing to sacrifice efficiency can we do it in a way which is relatively simple to describe? Yes, but there is one way in which this parser will be unsatisfactory.

Recognisers are easier to construct and understand than parsers so we'll start with a recogniser.

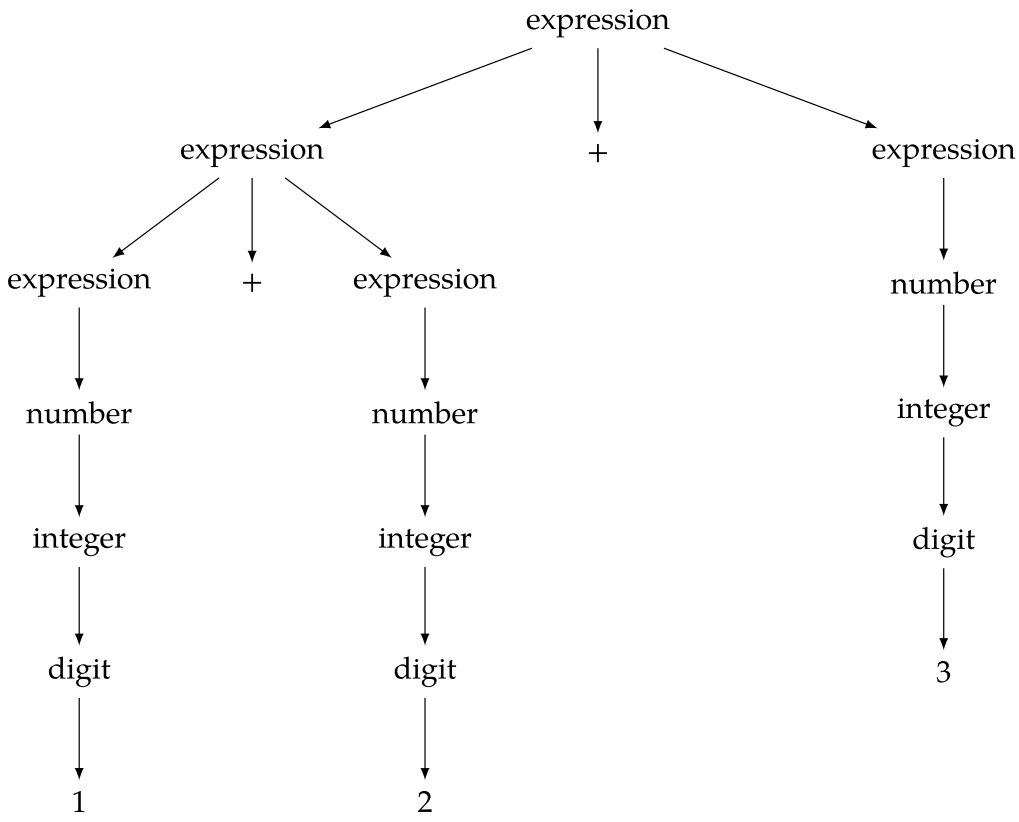


Figure 17: Full parse tree for 1 + 2 + 3

It's helpful to think in terms of nondeterministic computation. Normally we expect an algorithm to tell us what to do at each stage. Our grammar rules are like an algorithm in that we proceed by steps from a well defined initial state, one where we have a list consisting of just the start symbol. Each step takes the first symbol from the list and replaces it with one of the tokens corresponding to that symbol or expands it into one of the lists of symbols on the right hand side of a grammar rule for that symbol, depending on whether it's terminal or nonterminal. If we expanded a terminal to a token we check whether that token matches the next input token. If it does then we remove it and if it doesn't, either because there are no input tokens left or because the next one is different from the one we chose, then the computation terminates unsuccessfully. If we ever reach a point where there are no symbols left to process we check whether there are any input tokens left. If there are then the computation terminates unsuccessfully. If not then it terminates successfully.

Successful termination in the description above means that the input is recognised as valid, i.e. that it can be generated by the given grammar. Unsuccessful termination doesn't mean that the input can't be generated though, merely that it wasn't generated by the particular choices we made. There might be other choices which would generate it.

This is a recogniser rather than a parser but it's not too difficult to convert it into a parser. Each step in the algorithm processes a symbol and we just need to put this symbol in the correct position in the parse tree, which is the root in the case of the start symbol or as the child of whichever symbol we expanded to get it in the case of all the other symbols.

There's a trick to turn nondeterministic computations into deterministic ones. Instead of making any particular choice at each step we make all of them. More precisely, starting from the initial state we write down all states we can reach in a single step. Then we write down all states which can be reached from one of those states, also recording the path that led us to those states. Then we write down all the ones we could reach in a single step from those, again recording the path that led to each one. Whenever we write down a state we check whether it satisfies the terminating condition. If so then we check whether the computation terminated successfully or unsuccessfully. If it terminated successfully then we're done. We have the full path which led us to that state. We're in the same situation we would

be in if we had a "lucky guesser", who made the optimal choice at each stage, except that it will have taken us longer to get there. If we're in one of the unsuccessful terminating states then we don't need to, and indeed can't, continue looking for continuations of that computational path but we can consider continuations from the other states on our list, if there are any. Only if all of our paths reach a dead end does the computation terminate unsuccessful. Typically it doesn't terminate at all though.

This algorithm can conveniently by represented by a tree, with the initial state at the root and nodes for each possible computational path and arrows from each of those to its one-step continuations, which implies branching at each node where there are multiple choices for the next step. Unless we specify an upper bound on the number of steps this tree could well be infinite. It is for the parsing problem we just considered, which is why I won't attempt to draw the tree.

Does this work? That depends on whether the available choices at each stage are finite and also what you mean by work. If there are only finitely many choices available in each state and there is a solution, i.e. a computational path which terminates successfully then this method will find it. In fact the method can be modified to cope with an infinite variety of choices, as long as it's not too infinite. What the method can't be relied on for is to tell us when there is no solution. It could tell us, if all paths have reached a dead end. It's certainly possible though that there is no solution but there's always something else to try so the algorithm will just run forever.

For the parsing problem you should not do this. There are algorithms which are much faster and which are guaranteed to tell you when the problem has no solution, i.e. when the list of tokens which is your input does not belong to the language. You should use one of those instead. They aren't covered in this module though. Still, the idea of nondeterministic computation is one which we will meet again in this module. It's not always this useless.

Formal definition

Let LA be the set of lists all of whose elements are in A. We'll define this notion more precisely later but for now it suffices to note that lists are required to be of finite length, but could be of length 0. The set A is called the

alphabet of the language and its elements are called tokens. Any subset of LA is called a language.

This definition of language is broad enough to include a wide variety of meanings which are commonly given to the word, including

- programming languages like C, Python, Rust, LISP, Haskell, etc.
- data description languages like (parts of) SQL,
- file formats like .csv or .ini,
- specialised single purpose languages like `printcap` config file entry syntax,
- languages for mathematical logic like the ones we'll use for zeroeth and first order logic.

It may not always be clear which category a language belongs to. In the introduction I introduced a single purpose language for module enrollment but it turns out to be equivalent to one of the languages used for mathematical logic, namely that of the propositional calculus. Similarly you might think of PostScript as a single purpose language for page description but it is also a full programming language capable of anything any other programming language is capable of. I've written PostScript code to solve ordinary differential equations and to compose Lorentz transformations. This isn't as bizarre a thing to do as it might seem. If your aim is to produce nice diagrams and you have a language which can describe diagrams in a way every modern printer can understand and which is also a full programming language then why wouldn't you just do everything in that language? The answer to that question, as it turns out, is that debugging PostScript code is very painful.

The definition above doesn't really include natural languages, like English, Irish, Arabic, Japanese, or Toki Pona, used by humans for communicating for other humans. For those it's often unclear whether particular lists of tokens are valid elements of the language. Subsets of natural languages are often used for communication between humans and computers though. The subset of a natural language that a given computer programme emits is almost always a language by the definition above. The subset it accepts is always one as well. Also, many of the concepts described below were first

developed in the context of natural languages and only later was it noticed that they apply even better to languages used by computers.

Grammars

Some, but not all languages are describable by a grammar. Languages which are describable by a grammar are typically describable by more than one grammar. According to the definition above the language is the set of lists of tokens, not any particular way of describing which lists belong to the subset.

Here we mean, by the term grammar, a finite set of grammar rules which describe how more complicated expressions are built up from simpler ones. What we've considered so far are context free grammars, which always replace a single symbol by a list of zero or more symbols. More complicated grammar rules might allow the replacement of one list of symbols with another list of symbols. That takes us into the world of context sensitive grammars, a world you are well advised to avoid if possible.

Normally we are only interested in a language if its lists of tokens have some sort of interpretation, but it's important to understand that that's not part of either the language or the grammar. In linguistic terms, we're currently discussing only syntax, not semantics.

Hierarchy

The definition of language given above is deliberately very broad, but it is really too broad to be useful. In this it is similar to notions like binary relation or binary operation discussed earlier. Practically useful examples have more structure. As in abstract algebra, there is a hierarchy of levels of structure. The main levels of this hierarchy, from most restrictive to least, are

1. finite
2. regular
3. deterministic context free
4. context free

5. context sensitive
6. recursive
7. recursively enumerable
8. general

The easiest of these to define are finite, which just means a finite set of lists of tokens, and general, which is any set of lists of tokens. The levels in between have more complicated definitions, but are more useful.

Each level in the hierarchy above includes all the levels further on in the list, so every finite language is regular, every regular language is context free, etc. The step which is most likely to cause confusion is that every context free language is context sensitive. "Context sensitive" doesn't really mean that the language is sensitive to context, merely that it could be, while context free means that it definitely isn't.

This sort of terminology is often used in mathematics. In the theory of linear equations we make a distinction between homogeneous equations and inhomogeneous equations. Homogeneous equations have zero constant term. Inhomogeneous equations aren't required to have zero constant term but are certainly allowed to. This means that every homogeneous equation is inhomogeneous. That certainly sounds weird but we define things in this way because there's simply nothing of interest to be said about equations whose constant term is non-zero which doesn't apply equally well when the constant term is zero. Similarly, the class of languages which are context sensitive but not context free simply has no interesting properties and therefore isn't worth naming.

There are a number of different, but equivalent, ways to describe the various levels in this hierarchy. We'll discuss this in more detail in later chapters, but I'll just mention here that one of these is in terms of the types of idealised machines which can recognise them. Regular languages, for example, turn out to be precisely those for which it's possible to construct a finite state automaton recogniser. As an illustration, the figure shows a finite state automaton which recognises the language of integer multiples of 3, which I gave a grammar for earlier.

We could, of course, ask the same question for other languages, like the language of palindromes. Is it possible to construct a finite state automaton

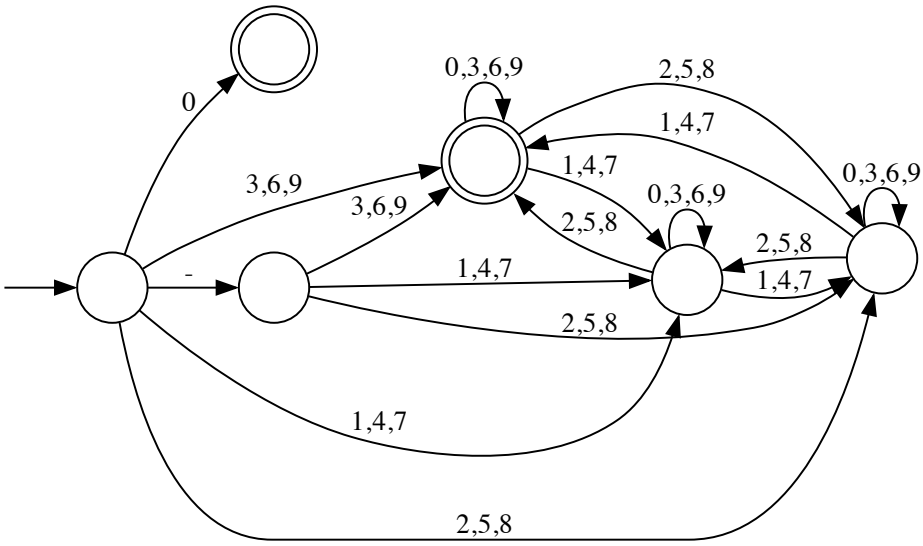


Figure 18: A finite state automaton recogniser

which recognises the language of palindromes? The answer turns out to be no, but how could we possibly prove this? Writing down all finite state automata and checking that none of them work is not an option, both because there are infinitely many of them and because it's not even obvious how we would check whether a single automaton recognises the language. We'll return to this question in a later chapter.

Other levels in the language hierarchy can also be described in terms of the types of idealised machines which can recognise them. The most important of these is the recursive enumerable languages, recognised by Turing machines. Another important class is the context free languages, recognised by pushdown automata. A pushdown automaton is essentially a finite state automaton which has a stack available for storage. A Turing machine can be thought of as a finite state automaton with a pair of stacks.

A good rule of thumb when developing a language for a specific purpose is to choose one as early as possible in the list above, and to describe it by a grammar at that level, or not much higher. One of the reasons for this is that it makes automated processing of the language much easier. Most modern programming languages are technically context sensitive, but try to segregate their context sensitive features as much as possible. The re-

mainder is context free, with significant parts which are regular or even finite. When you think of automated processing of a programming language your first thought is likely to be of an interpreter or compiler but in fact there are many other programs which process programming languages. Most modern text editors, for example, offer syntax colouring, which means parsing the source code of program in order to colour tokens according to the terminal they belong to.

Back to the beginning

In the introduction I informally introduced a language for module selection rules. I can now provide an actual grammatical description. In fact I can provide more than one. The typical way to do things in practice would be with two stages, a lexical analyser and a parser. If the lexical analyser is doing the work of breaking the input into tokens then our grammar, in the same notation as before, is

```
statement  : statement2
           | statement "∨" statement2
statement2 : statement3
           | statement2 "∧" statement3
statement3 : statement4
           | "¬" statement4
statement4 : MODULE
           | "(" statement ")"
```

The non-terminal symbols are the "∨", "∧", "¬", "(" and ")", each with that string as the only token belonging to the symbol, and `MODULE`, which is a symbol containing all possible module names. There are four different nonterminals, the four types of statements, and `statement` is the start symbol.

Separating out the input into tokens and assigning them to terminal symbols is, as usual, the job of the lexical analyser. In practice we tend to give the syntactic analyser another job as well, removing unnecessary white space, i.e. space characters, tabs, and new lines. This allows us to write more readable input without burdening the parser.

In fact even simpler grammars are possible but this one is unambiguous

and always generates the correct parse tree. The different levels of statements ensure this. For example, "¬ Algebra ∧ Geometry" will be parsed as if it were "((¬ Algebra) ∧ Geometry" rather than as "(¬ (Algebra ∧ Geometry))" because "¬" can only appear before a level 4 statement. "Algebra", as a module name, is a level 4 statement but "Algebra ∧ Geometry is a level 2 statement.

This language is simple enough that we could dispense with the separate lexical analysis step entirely though and work directly with characters rather than strings as tokens. A grammar which does this is

```
statement  : statement2 | statement spaces "∨" spaces statement2
statement2 : statement3 | statement2 spaces "∧" spaces statement3
statement3 : statement4 | "¬" spaces statement4
statement4 : module | "(" statement ")" | "(" spaces statement ")"
       | "(" statement spaces ")" | "(" spaces statement spaces ")"
ows        : | spaces
spaces     : " " | spaces " "
module     : words
words      : word | words spaces word
word       : letter | letters letter
letter     : "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
           | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R"
           | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "0"
           | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
           | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
           | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
           | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
```

`ows` is an abbreviation for "optional white space". For simplicity I've assumed that this consists solely of spaces, not tabs or line breaks, and the module names don't use any characters other than English letters or numbers.

Zeroeth order logic

Formal vs informal proof

In the twenty three centuries since Euclid mathematical proofs have gradually become more formalised. The ultimate step in formalisation is proofs which can be, and indeed are, checked entirely mechanically.

There are a few advantages to such proofs. Informal proofs rely on intuition. Intuition is often wrong. More subtly, it is often correct, but only on a particular interpretation. But theories, if formulated sufficiently generally, may admit multiple interpretations. This can be quite useful. For example, much of elementary algebra works equally well regardless of whether the numbers in question are rational, real or complex. Mechanically checked formal proofs can ensure that conclusions of theorems follow from their hypotheses under any interpretation which is consistent with the axioms and rules of inference, not just a particular interpretation. They will continue then to hold under interpretations which would never have been considered by the original writer and readers of a proof. Projective geometry, for example, was originally developed for perspective drawing, but is now principally used in settings like error-correcting codes with "lines" and a "plane" with only finitely many points. This is possible because the theory was formulated in a way which didn't exclude this unanticipated interpretation, and proofs were given for the major theorems which did not rely on any intuition that lines or planes must be infinite.

There are, however, three disadvantages to completely formalised mathematics.

The first disadvantage is that such arguments are hard for humans to read. There is simply too much detail. The language required to remove all ambiguities is too unfamiliar. It is too difficult to identify the important steps. Here, for example, is a formal proof that two times two equals four, in a

formal system we will encounter later.

1. $\{\forall x.[(x \cdot 0) = 0]\}$
2. $[(0'' \cdot 0) = 0]$
3. $[(0'' \cdot 0)' = 0']$
4. $[(0'' \cdot 0)'' = 0'']$
5. $[\forall x.(\forall y.\{(x \cdot y') = [(x \cdot y) + x]\})]$
6. $(\forall y.\{(0'' \cdot y') = [(0'' \cdot y) + 0'']\})$
7. $\{(0'' \cdot 0') = [(0'' \cdot 0) + 0'']\}$
8. $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
9. $(\forall y.\{[(0'' \cdot 0) + y'] = [(0'' \cdot 0) + y]'\})$
10. $\{[(0'' \cdot 0) + 0''] = [(0'' \cdot 0) + 0']'\})$
11. $\{(0'' \cdot 0') = [(0'' \cdot 0) + 0']'\}$
12. $\{[(0'' \cdot 0) + 0'] = [(0'' \cdot 0) + 0]'\}$
13. $(\forall x.\{[x + 0] = x\})$
14. $\{[(0'' \cdot 0) + 0] = (0'' \cdot 0)\}$
15. $\{[(0'' \cdot 0) + 0] = 0\}$
16. $\{[(0'' \cdot 0) + 0]' = 0'\}$
17. $\{[(0'' \cdot 0) + 0'] = 0'\}$
18. $\{[(0'' \cdot 0) + 0']' = 0''\}$
19. $\{[(0'' \cdot 0) + 0''] = 0''\}$
20. $[(0'' \cdot 0') = 0'']$
21. $\{(0'' \cdot 0'') = [(0'' \cdot 0') + 0'']\}$
22. $(\forall y.\{[(0'' \cdot 0') + y'] = [(0'' \cdot 0') + y]'\})$
23. $\{[(0'' \cdot 0') + 0''] = [(0'' \cdot 0') + 0']'\}$
24. $\{(0'' \cdot 0'') = [(0'' \cdot 0') + 0']'\}$
25. $\{[(0'' \cdot 0') + 0'] = [(0'' \cdot 0') + 0]'\})$
26. $\{[(0'' \cdot 0') + 0] = (0'' \cdot 0')\}$
27. $\{[(0'' \cdot 0') + 0] = 0''\}$
28. $\{[(0'' \cdot 0') + 0]' = 0'''\}$
29. $\{[(0'' \cdot 0') + 0'] = 0'''\}$
30. $\{[(0'' \cdot 0') + 0']' = 0''''\}$
31. $[(0'' \cdot 0'') = 0'''']$

The second difficulty with formal proofs is that we need to know that the formal system in which we are working, and the proof checker we use to verify the proofs, are correctly designed. In order to have any confidence in the results, this needs to be proved, but how? We can give an informal proof, or we can give a formal proof, but this would have to be done in another formal system, and checked by a different proof checker, since the correctness of this one has yet to be established. So eventually even the most formal of proofs has to be based on an informal foundation. The gain from using formal systems is therefore not in getting rid of all appeals to human intuition, but rather in reducing those to a tightly defined core. There is also the matter of checking that the formal statements being proved have the desired meaning under the interpretation we've adopted for statements in the system, which is in fact a frequent source of error.

The third difficulty with formal proofs is that they can't accomplish the purpose for which they were originally intended. It was originally hoped that one could find a formal system in which it would be possible to formulate and prove all true statements in mathematics, and of course only true statements, since a system which proves false statements is not of much

use. It's now known that this can't be accomplished even for arithmetic. Any system which is consistent, in the sense that it cannot be used to prove contradictions, will be incomplete, in the sense that not all true statements will be provable.

This doesn't mean that formal proofs are useless. The exercise of giving a formal proof that a piece of code works for all allowed inputs, for example, will almost always reveal that it doesn't. A large scale project was conducted in 2009 to show that the L4 microkernel was free of bugs, in the sense that it was proven to implement its design specification. The exercise uncovered a large number of previously unsuspected bugs, which were then fixed. Some of these were bugs in the implementation, but others were bugs in the specification itself, where assumptions which should have been explicit had been left unstated.

Formal systems

The preceding section referred to formal systems without defining them. A formal system consists of a formal language, a set of axioms and a set of rules of inference. It does not include an interpretation, although we're usually interested in a formal system because it admits at least one useful interpretation.

The language describes the elements from which statements are built and the grammatical rules which describe how they are built from those elements. A rule of inference describes how a statement can be derived from other statements. A proof in a formal system is a finite sequence of grammatically correct statements, each of which is either an axiom or is derived, in accordance with the rules of inference, from statements earlier in the sequence. A statement is called a theorem if it forms the final statement in such a sequence and that sequence is called a proof of the theorem.

The set of axioms can be empty, finite and non-empty, or infinite. All of these cases occur in commonly used systems. In principle the set of rules of inference can also be empty, finite and non-empty, or infinite, but systems with no rules of inference are uninteresting because the only theorems in such systems are the axioms. Systems with infinitely many rules of inference are not often used.

The rules of grammar and rules of inference are required to be not merely constructive, but analytic. It should be possible not just to build more complicated expressions from simpler expressions but also to analyse a complicated expression to determine uniquely how it was built up. This process should be purely mechanical, relying solely on the structure of the expression and not on any intended interpretation. Similarly the rules of inference should enable us not just to derive statements from other statements but to check that a statement is indeed derivable from earlier statements. If there are infinitely many axioms then it should be possible not just to generate axioms but to verify whether a statement is an axiom. These requirements force us to consider questions of computability in the description of formal languages.

A language for zeroeth order logic

The propositional calculus, often called zeroeth order logic, governs the use of logical operators like "and", "or" and "if ... then". It does not concern itself with quantifiers, like "for all" or "there exists", which belong to first order logic. It does not concern itself with the meaning of the statements combined with those connectives. It should be noted though that it can only be expected to behave as expected when those statements are either definitely true or false. It does not cope well with statements like "this statement is false."

Our language for zeroeth order logic will consist of variables and logical operators. The variables are Boolean variables and the logical operators are Boolean operators, but in this chapter we have no other type of variable or operator so I'll drop the word Boolean until this next chapter. We'll use lower case letters, starting with p for variables and single symbols for logical operators. In particular we'll use $\wedge$ for "and", $\vee$ for "or", and $\neg$ for "not". The grammar will also allow the use of $\supset$ for "implies", $\overline{\wedge}$ for "nand", $\veebar$ for "nor", $\equiv$ for "if and only if", $\not\equiv$ for "xor", and $\subset$ for "if", but we'll only ever use the first two of those, will use the second only briefly. We'll use an infix notation but we'll use a fully parenthesised version rather than relying on precedence and associativity rules. To make it easier to spot matching parentheses we'll use not just (and) but also [and] and { and }. These are to be regarded as fully equivalent though. Anywhere they appear in the following discussion any of the above pairs may be replaced with any

other. We'll use the lower case Latin letters p, q, r, s and u for variables. We skip t here to avoid confusion with the constant symbol t, meaning "true", which isn't part of our language, but which we well use, along with f for "false", in talking about the language.

The weird symbols for logical operators are unfamiliar at first sight but forcing all symbols to be single characters shortens formulae, makes a lexical analyser unnecessary and allows us to dispense with whitespace as a way of separating symbols.

For theoretical purposes it's convenient to allow an infinite number of variables so we'll also allow adding arbitrarily many exclamation points to these letters to create new variables, like p, $p!$, $p!!$, etc. We'll never actually encounter an example where we run out of Latin letters though, so this will remain just a theoretical possibility. Some treatments of zeroeth order logic also have constants symbols for the values "true" and "false" but these won't be part of our language. The letters t and f will appear in various places though, like truth tables.

Our grammar is then

```
statement  : expression
expression : variable
       | "(" expression binop expression ")" | "(" "¬" expression ")"
       | "[" expression binop expression "]" | "[" "¬" expression "]"
       | "{" expression binop expression "}" | "{" "¬" expression "}"
variable   : letter | variable "!"
letter     : "p" | "q" | "r" | "s" | "u"
binop      : "∧" | "∨" | "⊃" | "⊼" | "⊻" | "≡" | "≢" | "⊂"
```

The spaces separate symbols in the specification. They aren't part of the language. Quoted strings are terminal symbols. The quotes are not part of the symbol.

`binop` is short for binary operator and includes the operators $\wedge$, $\vee$, $\supset$, $\barwedge$, $\veebar$, $\equiv$, $\not\equiv$, $\subset$.

This language is unambiguous. Why? Any expression longer than a single variable is bracketed by parentheses of some type. The rules which expand to such expressions all have a single Boolean operator joining one or two expressions. There may be more operators in those expressions, but they

are contained within their own set of parentheses. We can identify which rule (alternate) was used by looking at the type of parentheses and the one operator which is not further parenthesised. For example $[(\neg p) \wedge (q \vee r)]$ must have been generated by

`[ expression ∧ expression ]`

This allows top down parsing: start with the whole statement, figure out which rule generated it, figure out which rule generated the expression(s) within it, repeat until we reach the level of variables.

We could have taken `expression` as the start symbol and dispensed with `statement` entirely. It will be convenient to have the distinction when we talk about rules of inference though. Every `statement` is an `expression` but not every `expression` is a `statement`. When we discuss the rule of substitution, for example, it's important that when we substitute an expression for a variable in a statement that we replace all occurrences of that variable in the the statement, not just all in some particular expression occurring in the statement.

As an example of the language, consider the statement

$$\{[p \vee (q \vee r)] \supset [(p \vee q) \vee r]\}.$$

This has the abstract syntax tree given in the diagram.

There are certain symbols which are not part of our language but which we will use for talking about the language. t and f have been mentioned previously. We'll use the upper case Latin letters P, Q, R, S and U to stand for arbitrary expressions. This is particularly useful in stating rules of inference. One commonly used rule of inference, for example, says that from statements of the form P and $(P \supset Q)$ we can deduce the statement Q. Here any expression can be substituted for P and Q. P and Q themselves, though, do not belong to the language. It's understood, as discussed above, that different types of brackets are interchangeable so an instance of the rule above would be that from $(r \wedge s)$ and $[(r \wedge s) \supset (r \vee s)]$ we can deduce $(r \vee s)$. This saves us from needing to repeat each rule three times, once for each set of brackets.

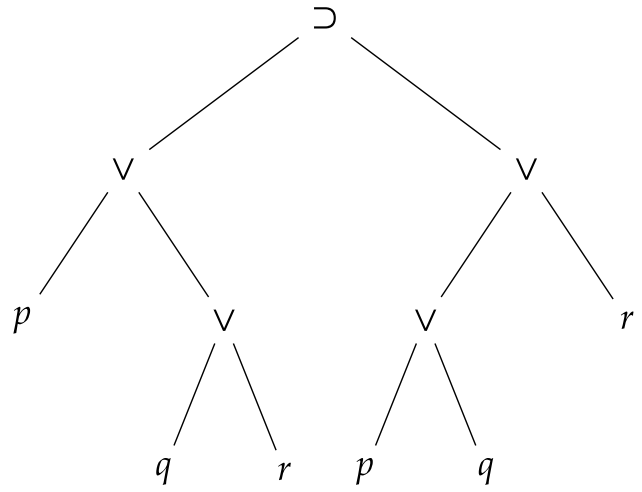


Figure 19: Syntax tree for $[p \vee (q \vee r)] \supset [(p \vee q) \vee r]$

Interpretation(s)

The standard interpretation is that the symbols "$\wedge$", "$\vee$", "$\neg$", and "$\supset$" for "and", "or", "not" and "implies" mean what you think they do, assuming you think "or" is always inclusive and you interpret "$\supset$" the way mathematicians and logicians do, i.e. that the expression is true if the hypothesis is false or the conclusion is true. As we discussed in the introduction $(P \supset Q)$ has the same meaning as $((\neg P) \vee Q)$. The meaning of "$\supset$" can sometimes confuse people. Under the interpretation above $[(p \supset q) \vee (q \supset p)]$ is true for any p and q. If this seems odd to you then you are probably thinking in terms of causality rather than logical implication.

Like "$\supset$" the more exotic symbols are all expressible in terms of "$\wedge$", "$\vee$", and "$\neg$". $(P \barwedge Q)$ has the same meaning as $(\neg(P \wedge Q))$. $(P \veebar Q)$ has the same meaning as $(\neg(P \vee Q))$. $(P \equiv Q)$ has the same meaning as $((P \wedge Q) \vee ((\neg P) \wedge (\neg Q)))$. $(P \not\equiv Q)$ has the same meaning as $((P \wedge (\neg Q)) \vee ((\neg P) \wedge Q))$. It's the exclusive or which we discussed earlier. $(P \subset Q)$ has the same meaning as $(P \vee (\neg Q))$.

The variables are Boolean variables. They can take the values true or false. Technically every possible assignment of values to the variables is a different interpretation of the language. Statements which are true in any

of these interpretations, i.e. for any assignment of truth values to the variables occurring in them, are called tautologies. Statements which are true in some interpretation, i.e. for some assignment of truth values to the variables, are said to be satisfiable. Note that it's only interpretations of the kind described above which are relevant. In judging whether a statement is a tautology or is satisfiable we don't consider, for example, interpretations where "$\vee$" means exclusive or.

Truth tables

Having assigned truth values to the variables we can work our way up to assign values to more and more complicated expressions. The way values are combined is summarised in "truth tables". The ones for the four basic operators are

P	Q	$(P \wedge Q)$
f	f	f
f	t	f
t	f	f
t	t	t

P	Q	$(P \vee Q)$
f	f	f
f	t	t
t	f	t
t	t	t

P	$(\neg P)$
f	t
t	f

P	Q	$(P \supset Q)$
f	f	t
f	t	t
t	f	f
t	t	t

I've written these with expressions P and Q rather than variables p and q because these can be applied to any expression in our language, not just to variables.

As an example of combining these to assign truth values to more complicated expressions consider the expression $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$.

We have

p	q	r	$(p \supset q)$	$(q \supset r)$	$[(p \supset q) \wedge (q \supset r)]$	$(p \supset r)$	$\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$
f	f	f	t	t	t	t	t
f	f	t	t	t	t	t	t
f	t	f	t	f	f	t	t
f	t	t	t	t	t	t	t
t	f	f	f	t	f	f	t
t	f	t	f	t	f	t	t
t	t	f	t	f	f	f	t
t	t	t	t	t	t	t	t

So the expression $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ evaluates as true no matter what truth values are assigned to p, q and r. In the terminology introduced earlier it is a tautology.

As another example, $\{[p \vee (q \vee r)] \supset [(p \vee q) \vee r]\}$ is also a tautology, as shown by the following truth table

p	q	r	$(q \vee r)$	$[p \vee (q \vee r)]$	$(p \vee q)$	$[(p \vee q) \vee r]$	$\{[p \vee (q \vee r)] \supset [(p \vee q) \vee r]\}$
f	f	f	f	f	f	f	t
f	f	t	t	t	f	t	t
f	t	f	t	t	t	t	t
f	t	t	t	t	t	t	t
t	f	f	f	t	t	t	t
t	f	t	t	t	t	t	t
t	t	f	t	t	t	t	t
t	t	t	t	t	t	t	t

We can think of $\{[p \vee (q \vee r)] \supset [(p \vee q) \vee r]\}$ as expressing the fact that $\vee$ is an associative operator, so we shouldn't be surprised to find that it is a tautology.

The fact that truth tables apply to expressions as well as variables has an important consequence. If a statement in the language is a tautology, i.e. is true for all possible values of the variables, then it must remain a tautology when any expressions are substituted in for those variables. This is called

the "Rule of Substitution" and a statement obtained in this way is called a "substitution instance" of the tautology we started with. It is commonly used as a rule of inference in formal systems for zeroeth order logic. Using the rule of substitution we can see that since $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ is a tautology so is $\{[(P \supset Q) \wedge (Q \supset R)] \supset (P \supset R)\}$ for any expressions P, Q and R.

Informal proofs in zeroeth order logic

At the moment we have a language and an interpretation, or rather a class of interpretations of that language but we don't have the axioms or rules of inference necessary for a formal system so we can't do formal proofs. We can still do informal proofs though since our interpretation, or rather interpretations, give us a notion of truth. One method of informal proof is truth tables. It's not a very efficient method though. The number of logical operators appearing in an expression is called the "degree" of the expression. A truth table for an expression of degree d with n variables will have $d + n$ columns and 2^n rows. There are better methods, including what's called the "method of analytic tableaux'', which is our next topic.

The method of analytic tableaux is really just a bookkeeping device for proof by contradiction combined with a form of case by case analysis. Truth table methods also involve a form of case by case analysis, but analytic tableaux use a less drastic one, where the number of cases to consider needn't grow exponentially with the number of variables. I'll illustrate this with a few examples, first giving a version without tableaux and then showing how tableaux can be used to organise the arguments.

Our first example will be the tautology $\{[(\neg p) \wedge (\neg q)] \supset [\neg(p \vee q)]\}$. How could $\{[(\neg p) \wedge (\neg q)] \supset [\neg(p \vee q)]\}$ fail to be true? It's a statement of the form $(P \supset Q)$, where P is $[(\neg p) \wedge (\neg q)]$ and Q is $[\neg(p \vee q)]$. Strictly speaking it's of the form $\{P \supset Q\}$, but our convention is that different types of parentheses are considered interchangeable, so we cat treat it just like we treat statements of the form $(P \supset Q)$. We can tell that it's a $\supset$ type expression because $\supset$ is the only operator inside only one set of parentheses. For $(P \supset Q)$ to be false P would need to be true and Q would need to be false. In our case, $[(\neg p) \wedge (\neg q)]$ needs to be true and $[\neg(p \vee q)]$ needs to be false. We "want" $[(\neg p) \wedge (\neg q)]$ to be true and

$[\neg(p \vee q)]$ to be false. I've put the quotation marks in because ultimately we want a contradiction, so we will want the opposite of what I've just written, but for now we proceed as if we were trying to make this true. If $[(\neg p) \wedge (\neg q)]$ is true then so are $(\neg p)$ and $(\neg q)$. So both p and q are false. If $[\neg(p \vee q)]$ is false then $(p \vee q)$ is true. Then p is true or q is true. But we already know both are false. So $\{[(\neg p) \wedge (\neg q)] \supset [\neg(p \vee q)]\}$ can't fail to be true. In other words, it's a tautology. Is this any faster than writing down a truth table? Probably not, but it generalises better.

As a second example, consider the tautology $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$. Suppose $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ were false for some value of p, q and r. For those values $[(p \supset q) \wedge (q \supset r)]$ must be true and $(p \supset r)$ must be false. Since $[(p \supset q) \wedge (q \supset r)]$ is true so are $(p \supset q)$ and $(q \supset r)$. So in our hypothetical example $(p \supset q)$ and $(q \supset r)$ are true and $(p \supset r)$ is false. Since it is false p must be true and r must be false. This is as far as we can get without splitting the argument into cases. Since $(p \supset q)$ is true p is false or q is true. But we already saw that p is true so we can exclude that possibility and conclude that q must be true. Since $(q \supset r)$ is true q is false or r is true. But we already saw that q is true r is false so we can exclude both possibilities. Thus the assumption that there are p, q, and r which make $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ false is untenable. In other words $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ holds for all values of p, q, and r.

As a third example, consider $\{[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]\}$. Again we have something of the form $(P \supset Q)$, so for this to be false P must be true and Q must be false, i.e. $[p \supset (q \supset r)]$ is true and $[(p \supset q) \supset (p \supset r)]$ is false. $[(p \supset q) \supset (p \supset r)]$ is also of the form $(P \supset Q)$. For it to be false we need $(p \supset q)$ to be true and $(p \supset r)$ to be false. $(p \supset r)$ is also of the form form $(P \supset Q)$. For it to be false we need p to be true and r to be false. So $[p \supset (q \supset r)]$, $(p \supset q)$ and p are all true and r is false. How can $[p \supset (q \supset r)]$ be true? It's of the form $(P \supset Q)$ and so can be true if P is false or Q is true. In this case that means p is false or $(q \supset r)$ is true. But p is true so $(q \supset r)$ must be true.

Where are we? We wanted to show $\{[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]\}$ is true, so we assumed it was false and have found that $(q \supset r)$, $(p \supset q)$ and p must be true and r must be false. If $(p \supset q)$ is true then p is false or q is true, but p is true so q is true. If $(q \supset r)$ is true then q is false or r is true, but q is true and r is false, so we have a contradiction. There-

fore $\{[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]\}$ must be true. The main problem with such arguments is keeping track of what we know and, if we need to split things into cases, which case or subcase we're in.

Analytic tableaux

It can be difficult in arguments like the ones above to keep track of what's known and what isn't at each point in the argument. In fact the arguments above weren't too bad since on the two occasions we had to split the argument into cases we were immediately able to rule out one or both. We aren't always so fortunate.

There are several versions of tableaux. I'll use a version where we write true statements to the left of a vertical line and false statements to the right of it. We use existing statements to fill in more and more lines until we reach a point where we need to split into two cases. Then we'll draw diagonal lines down to a new pair of vertical lines, one for each case, and proceed in the same way with each of them. These are called branches. We can close off a branch whenever we have a statement which appears on both the left and right hand side of a vertical line, i.e. a statement which is both true and false, therefore a contradiction. We proceed in this way until all branches are closed or until we've explored all possible consequences of all statements in all branches.

The tableau corresponding to $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ is given in the accompanying figure.

The contradictions which allow us to close off branches are indicated by underlining the expression which has previously appeared on the other side of the vertical line.

As another example, we can show that $\{[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]\}$ is a tautology

If you compare these to the case by case arguments given previously you can see that they are essentially the same.

As yet another example, we can use an analytic tableau to show that $\{[p \vee (q \vee r)] \supset [(p \vee q) \vee r]\}$ is a tautology.

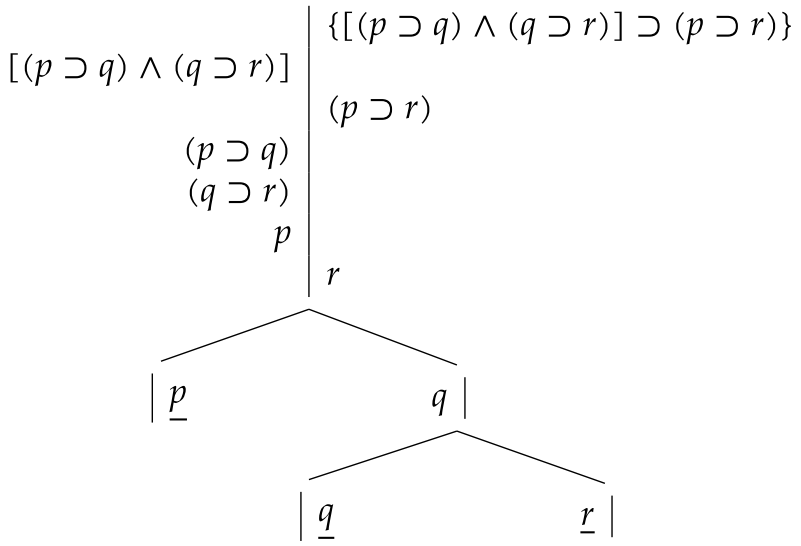


Figure 20: An analytic tableau for $[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)$

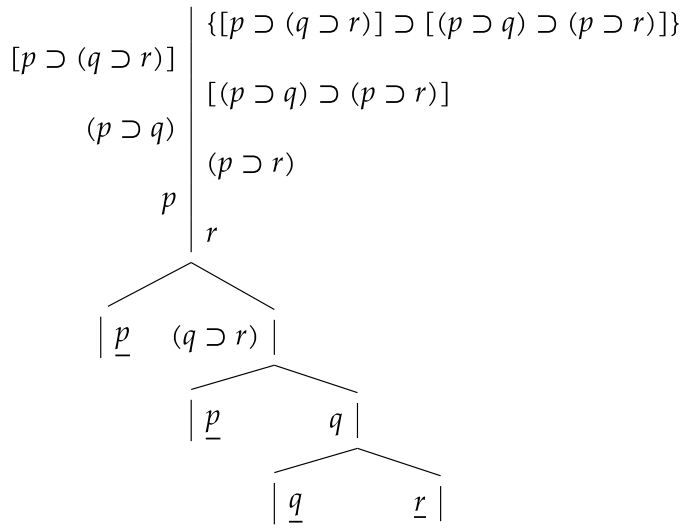


Figure 21: Analytic tableau for $[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]$

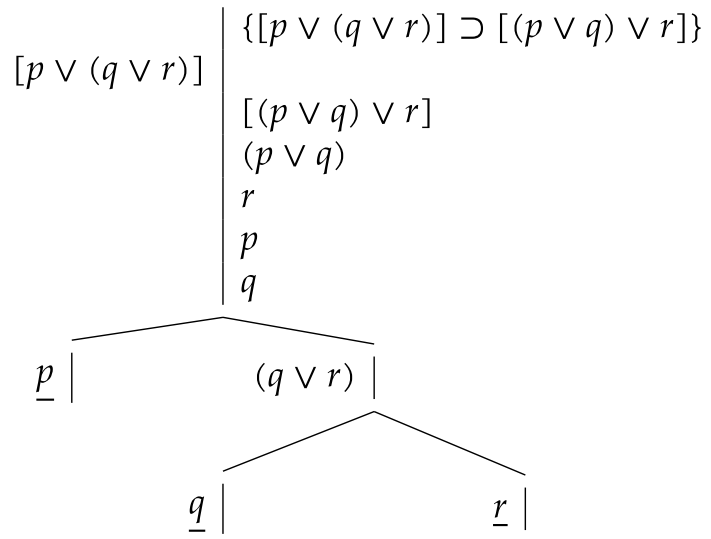
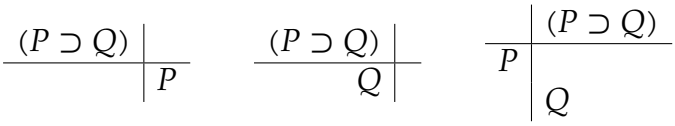


Figure 22: Analytic tableau for $[p \vee (q \vee r)] \supset [(p \vee q) \vee r]$

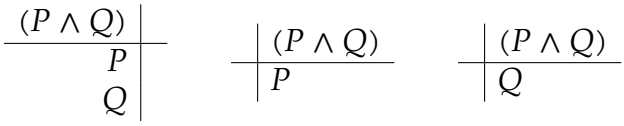
Tableau rules

All expressions in our language are built by joining simpler expressions with logical operators. For each operator there is a pair of tableau rules, one for the case where the expression appears to the left of of the vertical line, and one for the case where it appears on the right. We've met both of these for the operator $\supset$. When an expression of the form $(P \supset Q)$ appears to the left of the vertical bar the tableau branches into a branch with the P on the right of the bar and one with a Q on the left, reflecting the two ways $(P \supset Q)$ could be true, i.e. either P is false or Q is true. On the other hand when an expression of the form $(P \supset Q)$ appears to the right of the bar there is no branching. We get a P to the left of the bar and a Q to the right, reflecting the fact that $(P \supset Q)$ can be false only if P is true and Q is false. The standard way of depicting these rules is with diagrams. In addition to the vertical bar from earlier these diagrams have a horizontal bar. Above this horizontal bar is the statement whose consequences we're exploring and below the bar are those consequences, which are always one or the other of the subexpressions from which the expression was made, and which may appear on either side of the vertical bar. The diagrams for

$\supset$ are

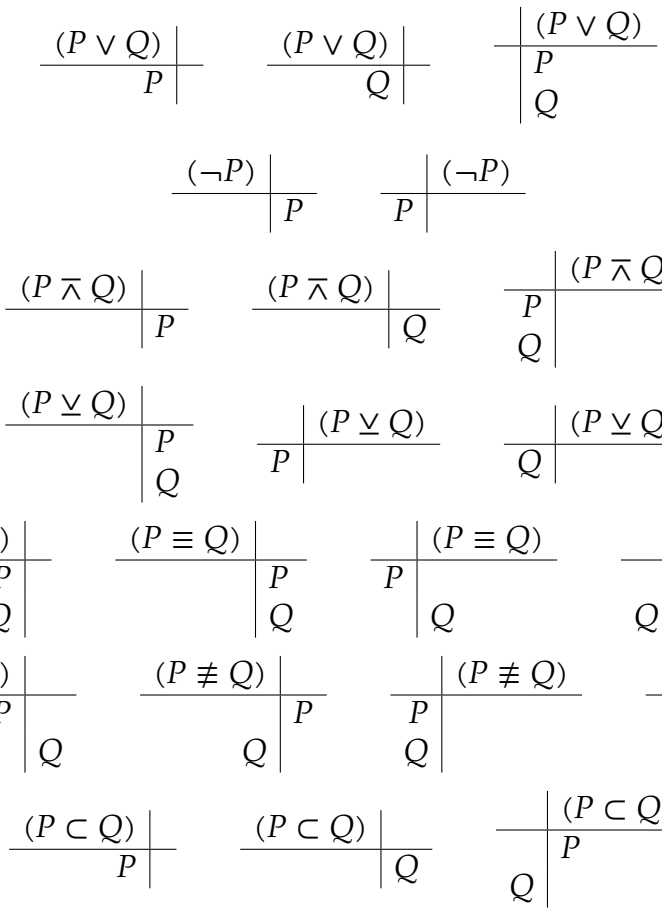


There are similar rules for $\wedge$.



The first of these rules appeared once in our example, when we split the expression $[(p \supset q) \wedge (q \supset r)]$ on the left hand side of the vertical line to a $(p \supset q)$ and a $(q \supset r)$, also on the left.

The diagrams for the remaining operators are



There is no need to memorise any of these. In each case you can reconstruct the diagram by asking yourself "How could this expression be true?" for the ones where it appears on the left and "How could this be false?" for the ones where it appears on the right.

Satisfiability

What happens if there's a branch you can't close? In other words, what happens if you've processed all consequences of all statements in the branch and have not found any statements which appear on both the left and the right of the line? In that case there is at least one choice of truth values which make all the statements on the left true and make all the statements on the right false. Finding such a choice is easy. You look for statements of degree zero, i.e. variables on their own without logical operators. Any which appear on the left are assigned the value true and any on the left are assigned the value false. Any which don't appear at all can be assigned either value. With these choices every statement of any degree on the left will be true and every statement of any degree on the right will be false.

Why does the method above work? Suppose it didn't. Then there would be a statement of lowest degree which is assigned the wrong value. Because of the way our grammar is defined this statement is constructed by applying a logical operator to statements of lower degree. These statements will appear on either the left or the right hand side of the vertical line lower down and, because they are of lower degree, they will have have been assigned the correct truth value.

Similarly, if you start with an expression on the left hand side of the vertical bar and can't close a branch then that means there are values of the variables for which the expression is true, i.e. that it is satisfiable. Not only do such values exist but you can find them by assigning variables which appear on the left the value true and variables which appear on the right the value false.

Examples

Consider the statement $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$. Is it a tautology, i.e. true for all values of p and q? Is it satisfiable, i.e. true for some values p and q? Is it neither?

To check whether it's a tautology we start a tableau with the expression $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ to the right of the bar and then apply our various rules.

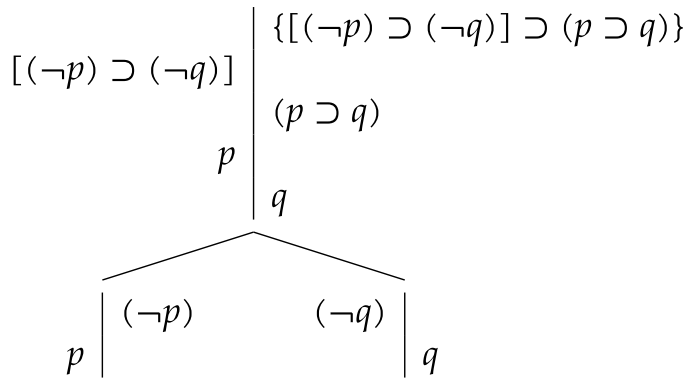


Figure 23: Checking whether $[(\neg p) \supset (\neg q)] \supset (p \supset q)$ is a tautology

All rules which can be applied have been applied and we can't close either of the two branches which were created by splitting the statement $[(\neg p) \supset (\neg q)]$ so $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is not a tautology. But the tableau tells us more than this. We can pick an open branch, for example the left branch, and look at which variables appear to the left and right of the bar. In this case p is on the left and q is on the right so taking p to be true and q to be false must make the statement $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ false. In this case we would have got the same values for p and q by choosing the other open branch, but that's an accident of this particular statement.

We don't, strictly speaking, need to check that assigning true to p and false to q makes $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ false but we certainly can. If you want to convince someone that $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is not a tautology, and therefore cannot be a theorem, it suffices to provide them with this counterexample. There's no need to show them the whole tableau and explain its meaning since they can check the value of the statement for these particular values and verify for themselves that it's false.

At this point we know that $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ for some values of p and q but we don't yet know whether it's true for other values of p and q. In other words, we don't yet know whether it is satisfiable. To check this

we start another tableau, this time with $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ to the left of the vertical bar.

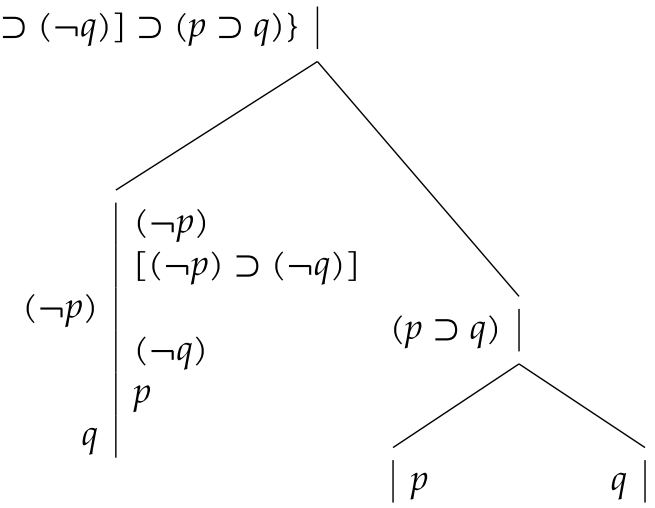


Figure 24: Checking whether $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is satisfiable

Once again we weren't able to close any branches. It wouldn't have mattered if we were able to close some. As long as there is one open branch the statement is satisfiable. We can find values of p and q which make the statement true by looking at where the variables are relative to the vertical bar on any open branch. If we take the rightmost branch, for example, then q appears to the left and p doesn't appear at all. We can therefore take q to be true and take either value for q. For definiteness we'll take it to be true as well.

We don't need to check that this works but we certainly can. More importantly, so can anyone else, so to convince someone that the statement $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is satisfiable it suffices to give them the example where p and q are both true. In particular, there's no need to explain the tableau method or convince them that it's correct, since they can just check the given example.

In this case we would have got a different example from choosing a different branch. Had we chosen the leftmost branch we would have got the example where p is false and q is true.

Similarly, we can use a pair of tableaux to show that the statement $[(p \supset q) \supset (q \supset p)]$ is satisfiable but is not a tautology. To show that it's satisfiable we start a tableau with it on the left.

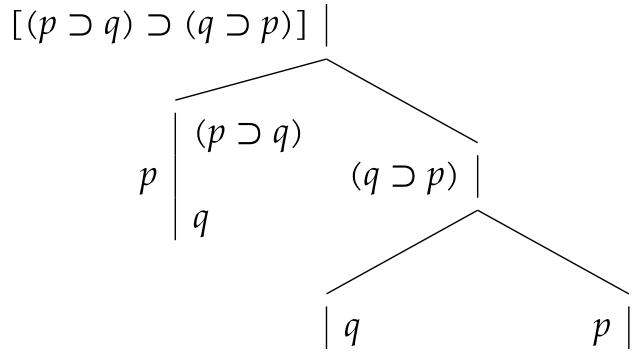


Figure 25: Checking that $[(\neg p) \supset (\neg q)] \supset (p \supset q)$ is satisfiable

There's nothing further to be done and at least one branch has failed to close–in fact none of them have, but we only need one–so the statement is satisfiable.

To show that it is not a tautology we start a tableau with the same statement on the right.

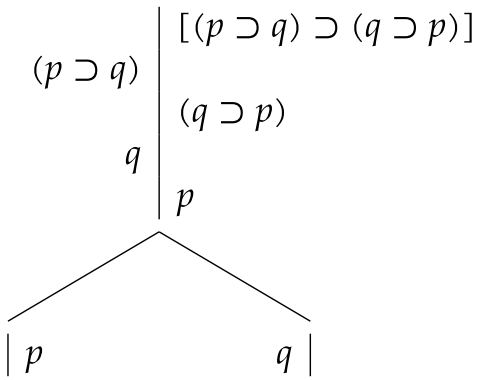


Figure 26: Checking $[(\neg p) \supset (\neg q)] \supset (p \supset q)$ isn't tautological

Once again there's nothing further to be done and we have a branch which hasn't closed. In fact there's more than one but again we only need one.

Since we started with our statement on the right and the tableau didn't close it must not have been a tautology.

Consequences

We can use the tableaux method to check whether a statement is a consequence of a list of other statements as well. We just put it to the right of the vertical bar and those statements to the left and fill in the tableau as before. If all branches close then it is indeed a consequence. If not then by choosing an open branch and looking at which side of the bar each variable lies on we can find truth values for them which cause the premises to be true and the purported consequence to be false. Our earlier method of proving, or disproving, tautologies can be viewed as a special case since a statement is a tautology if and only if it is a consequence of the empty list of statements.

Axiomatic systems for zeroeth order logic

The Nicod formal system

Perhaps the simplest formal system for zeroeth order logic is the Nicod system. As its language it uses the subset of our language for zeroeth order logic consisting of those lists where $\overline{\wedge}$, whose truth table is

P	Q	$(P \mathbin{\overline{\wedge}} Q)$
f	f	t
f	t	t
t	f	t
t	t	f

is the only logical operator appearing. There's no loss of expressiveness involved in this restriction since we can write $(P \wedge Q)$ as $[(P \mathbin{\overline{\wedge}} Q) \mathbin{\overline{\wedge}} (P \mathbin{\overline{\wedge}} Q)]$, $(P \vee Q)$ as $[(P \mathbin{\overline{\wedge}} P) \mathbin{\overline{\wedge}} (Q \mathbin{\overline{\wedge}} Q)]$, and $(\neg P)$ as $(P \mathbin{\overline{\wedge}} P)$. All the other operators were expressed in terms of $\wedge$, $\vee$ and $\neg$ so they can be expressed by first converting the expression into one involving those three operators and then converting them as above.

The Nicod system has a single axiom,

$$((p \mathbin{\overline{\wedge}} (q \mathbin{\overline{\wedge}} r)) \mathbin{\overline{\wedge}} ((s \mathbin{\overline{\wedge}} (s \mathbin{\overline{\wedge}} s)) \mathbin{\overline{\wedge}} ((u \mathbin{\overline{\wedge}} q) \mathbin{\overline{\wedge}} ((p \mathbin{\overline{\wedge}} u) \mathbin{\overline{\wedge}} (p \mathbin{\overline{\wedge}} u))))).$$

There is also only one rule of inference, that from two statements of the form P and $[P \barwedge (Q \barwedge R)]$ we can derive the statement R.

We can use truth table method to show that, no matter which of the 32 possible ways of assigning truth values to the variables p, q, r, s, and u we choose, the statement

$$((p \barwedge (q \barwedge r)) \barwedge ((s \barwedge (s \barwedge s)) \barwedge ((u \barwedge q) \barwedge ((p \barwedge u) \barwedge (p \barwedge u))))).$$

will always evaluate to true, i.e. that it is a tautology.

It is also possible to show that the rule of inference of the system has the property that if applied to tautologies it leads to a tautology. This follows from the truth table

P	Q	R	$(Q \barwedge R)$	$[P \barwedge (Q \barwedge R)]$
f	f	f	t	t
f	f	t	t	t
f	t	f	t	t
f	t	t	f	t
t	f	f	t	f
t	f	t	t	f
t	t	f	t	f
t	t	t	f	t

There is only one case in which both P and $[P \barwedge (Q \barwedge R)]$ are true and in that case R is also true.

It follows that any theorem must be a tautology, since we start from an axiom which is true in any interpretation which assigns to the operator $\barwedge$ the meaning described by its truth table given earlier and the rules of inference can only produce true statements from true statements. In other words any interpretation which assigns to the operator $\barwedge$ the meaning described by the truth table above and assigns any truth values whatever to its variables is a sound interpretation of the Nicod system. We say that a system is "sound" if the intended interpretation or interpretations are sound. The Nicod system is sound in this

We could add a second rule of inference, the rule of substitution discussed earlier. More explicitly, from any statement we can derive another statement by replacing each occurrence of one of the variables with an expression. In fact we can do this for each variable in the statement. Any proof

which uses the rule of substitution can be converted into one in the original system by repeating the proof of the statement into which we want to substitute, but making all of the substitutions everywhere in that proof. So the system with this extra rule has the same set of theorems as the original system. Rules like this are called derived rules of inference. Strictly speaking, whenever we us a derived rule of inference the result is a semi-formal proof rather than a formal proof.

The Łukasiewicz system

An alternative formal system for the propositional calculus is due to Łukasiewicz. It uses the subset of our general language for zeroeth order logic where the only logical operators are $\neg$ and $\supset$. There is no loss of expressiveness since $(P \wedge Q)$ has the same meaning as $\{\neg[P \supset (\neg Q)]\}$ and $(P \vee Q)$ has the same meaning as $[(\neg P) \supset Q]$.

The axioms are

$$[p \supset (q \supset p)],$$

$$\{[p \supset (q \supset r)] \supset [(p \supset q) \supset (p \supset r)]\},$$

and

$$\{[(\neg p) \supset (\neg q)] \supset (q \supset p)\}.$$

The rules of inference are the rule of substitution and a rule, known by the curious name of "modus ponens" which allows us to derive Q from P and $(P \supset Q)$.

The system as introduced by Łukasiewicz differs in one respect from that described above. Łukasiewicz used prefix notation in place of infix notation. He was, in fact, the first person to introduce prefix notation, and to notice that it allows one to dispense with parentheses. Łukasiewicz also used N and C in place of $\neg$ and $\supset$.

A direct proof of the soundness Łukasiewicz's system is slightly more complicated than a proof the soundness of Nicod's, because the system is larger and more complicated, but it can be done by the same method, using truth tables.

Because Łukasiewicz's system contains the $\neg$ operator we can also discuss consistency, which is the requirement that for any statement P at most one

of P and $(\neg P)$ is a theorem. In other words the system is free from contradictions. Unlike soundness, consistency is purely a property of the system, not the system and its interpretation. A small bit of interpretation is smuggled in because it's only the interpretation which tells us that the pair P and $(\neg P)$ form a contradiction but this is really the only aspect of the interpretation which is needed to discuss consistency. If you believe that P and $(\neg P)$ can't simultaneously be true then consistency follows from soundness because if they can't both be true then they can't both be tautologies and every theorem is a tautology.

For humans, proofs in Łukasiewicz's system are easier to read, write and check than in Nicod's system. This doesn't mean they are easy. Here is a proof of the theorem $(p \supset p)$, which we can easily check is a tautology by considering the two possible values of p:

1 $(p \supset (q \supset p))$
2 $((p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r)))$
3 $(p \supset ((q \supset p) \supset p))$
4 $((p \supset ((q \supset p) \supset p)) \supset ((p \supset (q \supset p)) \supset (p \supset p)))$
5 $(p \supset (q \supset p)) \supset (p \supset p))$
6 $(p \supset p)$

Statements 1 and 2 are axioms. Statement 3 follows from 1 by substituting $(q \supset p)$ for q. Statement 4 follows from 2 by substituting $(q \supset p)$ for q and p for r. Statement 5 follows from 3 and 4 by modus ponens. Statement 6 follows from 1 and 5 by modus ponens. More interesting theorems have, as you might expect, even longer proofs.

Proving the completeness of Łukasiewicz's system is easier than proving that of Nicod's, but I still won't do it.

Natural deduction

Some people find proving theorems in formal systems like Nicod's or Łukasiewicz's an entertaining sort of puzzle. Other people do not. What's undeniable is that such proofs have a very different flavour from those of the rest of mathematics. There was a reaction against these and similar axiomatic systems which led to what's known as "natural deduction".

One of the most important people behind this reaction was Łukasiewicz himself. Natural deduction systems are still formal systems, but their rules better reflect the way mathematicians typically think.

A formal system for natural deduction

There are a wide variety of natural deduction systems. We'll use one due to Douglas Hofstadter, with a few extra rules of inference. Having more rules of inference makes constructing proofs in the formal system easier, but it comes at the expense of making most proofs of statements about the formal system harder. The language of this system includes only the operators $\wedge$, $\vee$, $\neg$, and $\supset$. It has no axioms! In contrast it has a lot of rules of inference:

1. From statements P and Q we can deduce the statement $(P \wedge Q)$. Also, from any statement of the form $(P \wedge Q)$ we can deduce the statement P and the statement Q.
2. From the statement P we can deduce the statement $(P \vee Q)$, where Q is any expression.
3. From P and $(P \supset Q)$ we can deduce Q.
4. The expressions $[\neg(\neg P)]$ and P are freely interchangeable. In other words, anywhere an expression of one of these forms appears in a statement we may deduce the statement where it has been replaced by the other.
5. The expressions $(P \supset Q)$ and $[(\neg Q) \supset (\neg P)]$ are freely interchangeable.
6. The expressions $[(\neg P) \wedge (\neg Q)]$ and $[\neg(P \vee Q)]$ are freely interchangeable.
7. The expressions $[(\neg P) \vee (\neg Q)]$ and $[\neg(P \wedge Q)]$ are freely interchangeable.
8. The expressions $(P \vee Q)$ and $[(\neg P) \supset Q]$ are freely interchangeable, as are $(P \supset Q)$ and $[(\neg P) \vee Q]$
9. The "Rule of Fantasy", to be described below.
10. The "Rule of Substitution", subject to restrictions to be discussed below.

The first few rules specify the behaviour of the four logical operators. They are closely related to our tableaux rules. Half of the first rule, which is called the rule of joining and separation, can be thought of an equivalent

to the tableau rule

$$\begin{array}{r|l} (P \wedge Q) & \\ \hline P & \\ Q & \end{array}$$

for example. It reflects the fact that if $(P \wedge Q)$ is true then P and Q are true. It's important to remember though that that's a property of the intended interpretation, or rather interpretations, of the system, while the rule above is a rule of inference within the system. The third rule is one we've met before, under the name modus ponens. The fourth rule above is related to the tableau rule

$$\begin{array}{l|l} (\neg P) & \\ \hline & P \end{array} \qquad \begin{array}{l|l} & (\neg P) \\ \hline P & \end{array}$$

with the first one applied with $(\neg P)$ in place of P. It reflects the "fact" that if P is not not true then it is true. The quotation marks reflect the reality that not everyone accepts this a logical principle. This is one of the dividing lines between "classical" and "intuitionist" logic. It's tableau counterpart is more complicated. It consists of following both branches from a $(P \supset Q)$ but then using the P to immediately close off the left branch, which would have a P to the right of the bar.

The fifth rule is called the rule of the contrapositive, it is the basis of proofs by contradiction. It is another dividing line between "classical" and "intuitionist" logic. The sixth and seventh rules are two known as De Morgan's laws. The eighth rule of inference is really just the observation we made when discussing Łukasiewicz's system that $\vee$ is expressible in terms of $\neg$ and $\supset$.

Introducing and discharging hypotheses

The ninth rule, the "rule of fantasy" in Hofstadter's terminology, is more complicated to explain, but reflects a common practice in informal proofs. We often say "Suppose P". We then reason for a while under that assumption and reach a conclusion Q. We then conclude "So if P then Q." There are two common circumstances in which we do this. One is proof by contradiction, where we then immediately apply the rule of the contrapositive. The other is case by case analysis, which was the basis of our tableau method. In such applications there will be a separate application of the rule of fantasy for each possible case. Writers of informal proofs are under

no obligation to tell you which of these two uses they have in mind and sometimes you have to read the whole proof to find out but often a clue is is in the verb forms used. In a proof by contradiction people are more likely to write "Suppose ... were true" rather than "Suppose ... is true". But this is not something you can entirely rely on.

I've just described what the rule of fantasy is intended to do, but not the precise rules governing it. They're just a formalised version of the rules which mathematicians follow in informal arguments. We need some terminology. The step of saying "Suppose P" is called introducing the assumption or hypothesis P. The step of saying "So if P then Q" is called discharging this hypothesis or assumption. Everything in between is called the scope of the hypothesis. It's possible, and indeed common, to introduce further hypotheses within the scope of an existing one, and so have nested scopes. In arguments based on tableaux this corresponds to branches within branches.

Scope determines which statements are accessible for use by the other rules at any point in a proof. When you enter a new scope by introducing a hypothesis you retain access to everything in the scope you were in. When you leave that scope by discharging that hypothesis you lose access to all statements since you entered it. The only trace of any of the reasoning which took place within that scope is the single statement $(P \supset Q)$ generated by discharging the hypothesis. This restriction on the accessible statements is needed to ensure that you can't deduce a statement by introducing P as a hypothesis unless it's of the form $(P \supset Q)$. Otherwise you could prove all statements, true or false, by introducing them as a hypothesis and then using them outside of the scope of that hypothesis. All the other rules are to be interpreted as implicitly subject to this restriction. So when we say that from $(P \wedge Q)$ we can deduce P and Q we mean that in a scope where $(P \wedge Q)$ is accessible we can deduce P and Q. It doesn't allow us to deduce P or Q if the statement $(P \wedge Q)$ appeared after some hypothesis but has already been discharged.

Statements outside the scope of any hypotheses are said to have "global" scope. Only such statements are theorems.

In informal proofs it can be difficult to spot where hypotheses are introduced and where they're discharged and therefore difficult to know which ones are accessible. This is particularly problematic in proofs by contradiction. The whole point of a proof by contradiction is that the hypothesis

which is introduced will later be shown to be false. Everything in the scope of that hypothesis could, and usually does, depend on that false hypothesis and therefore should never be used outside that particular proof by contradiction argument. This is a common source of error for students. If you scan through a textbook looking for things which might be useful in problem you're attempting then you may find useful statements in the proof of a theorem. They'll typically depend on various hypotheses which have been introduced though and can't safely be used in a context where those hypotheses aren't known to be true.

In a formal system we need some way to indicate the introduction and discharging of hypotheses.

The safest way to do it would be to include a list of all active, i.e. not yet discharged, hypotheses before each statement. That solves the problem described above of using statements outside of their scope, but at the cost of making proofs very long and repetitive.

Jaśkowski, one of the founders of the theory of natural deduction, used boxes. A box encloses all the statements within a give scope and it's straightforward to see which statements are available within it. Starting from wherever we currently are we have available any statement we can reach by crossing zero or more box boundaries from inside to outside, but we are not allowed to cross any from outside to inside. This notational convention would probably have been more popular if it weren't a nightmare to typeset.

A popular alternative is to use indentation. Every time we introduce a hypothesis we increase the indentation and every time we discharge one we restore it to its previous value. The first statement after the indentation is increased is the newly introduced hypothesis. The first statement after the indentation has been restored is the result of discharging the hypothesis, i.e. the statement $(P \supset Q)$ where P is the hypothesis and Q is the last statement before the indentation was restored. This is a very compact notation but using spaces for indentation can cause problems. Screen readers will generally ignore spaces. Even for sighted readers judging the number of spaces at the start of a line is error-prone. It's better to use a non-whitespace character. We'll use dots.

The "rule of fantasy" is Hofstadter's terminology. It accurately reflects the

use of the rule, to explore the consequences of a statement not known to be true, but don't expect anyone to understand you if you use the term outside of this module.

There is some redundancy in the rules above, in the sense that there are proper subsets of those rules with the property that any statement which has a proof using the full set also has a proof using only the subset. But the point of a natural deduction system is to formalise something close to the way mathematicians actually reason rather than to have an absolutely minimal system. If you like minimal systems then you're better off with Nicod.

There are, as we'll see some restrictions needed on the rule of substitution but I'll get to those once we have some examples of proofs. In the interim I will be careful not to use that rule.

Some proofs

The shortest proof in this system is

$$\begin{array}{ll} 1 & . \quad p \\ 2 & (p \supset p) \end{array}$$

which introduces p as a hypothesis and immediately discharges it. The line numbering isn't technically part of the proof; I've just added it to make it easier to refer to individual lines. This proof shows that the statement $(p \supset p)$ is a theorem in this system. We saw that it was a theorem in Łukasiewicz's system as well, but with a considerably longer proof.

As another example, consider this proof of the statement $\{p \supset [q \supset (p \wedge q)]\}$.

$$\begin{array}{ll} 1 & . \quad p \\ 2 & . \quad . \quad q \\ 3 & . \quad . \quad (p \wedge q) \\ 4 & . \quad [q \supset (p \wedge q)] \\ 5 & \{p \supset [q \supset (p \wedge q)]\} \end{array}$$

Our first step is again to introduce a hypothesis. In this system the first step is always to introduce a hypothesis. There are no axioms and every other rule deduces a statement from previous statements, of which we have none.

It's not hard to guess which hypothesis to introduce. The statement we want to prove is $\{p \supset [q \supset (p \wedge q)]\}$ and it starts with $p \supset$ so if we introduce p and then manage to prove $[q \supset (p \wedge q)]$ then we will be done. So we introduce p. What next? We want to prove $[q \supset (p \wedge q)]$. It starts with $q \supset$ so try the same thing, introducing q as a further hypothesis. If we can prove $(p \wedge q)$ within the scope of this hypothesis then we will have proved $[q \supset (p \wedge q)]$ within the scope of the hypothesis p and therefore will have proved $\{p \supset [q \supset (p \wedge q)]\}$ within the global scope, i.e. in the absence of any hypotheses. At this point we have two statements available within our current scope p and q. We just introduced q. We inherited p from the outer scope. This is what I meant when I said that when you enter a new scope by introducing a hypothesis you retain access to everything in the scope you were in. So we have p and q and want $(p \wedge q)$. Fortunately our first rule of inference, the rule of joining and separation, does exactly this. So the proof may look strange at first but really at each stage we do the only thing we can and it works.

As another example, consider $[p \vee (\neg p)]$, which is often called the "law of the excluded middle". In this case it's less obvious how to start. For the reasons discussed above we must start by introducing a hypothesis, but what hypothesis? The statement isn't of the form $(P \supset Q)$ for any choice of expressions P and Q. Looking at our rules though we see that one of them allows us to derive $[p \vee (\neg p)]$ from $[(\neg p) \supset (\neg p)]$. That is easily proved with the fantasy rule. We introduce the hypothesis $(\neg p)$ and then immediately discharge it. The complete proof is

$$\begin{array}{ll} 1 & . \quad (\neg p) \\ 2 & [(\neg p) \supset (\neg p)] \\ 3 & [p \vee (\neg p)] \end{array}$$

A slightly more complicated example is the following proof of $\{[p \wedge (\neg p)] \supset q\}$. This time I'll just indicate which rule is used in

each line and not explain the strategy behind it.

1 . $[p \wedge (\neg p)]$
2 . p
3 . $(\neg p)$
4 . . $(\neg q)$
5 . . $[\neg(\neg p)]$
6 . $\{(\neg q) \supset [\neg(\neg p)]\}$
7 . $[(\neg p) \supset q]$
8 . q
9 $\{[p \wedge (\neg p)] \supset q\}$

The first line is the introduction of a hypothesis, also known as the rule of fantasy. The hypothesis is $[p \wedge (\neg p)]$. The second and third lines are from our first rule of inference, "From statements P and Q we can deduce the statement $(P \wedge Q)$. Also, from any statement of the form $(P \wedge Q)$ we can deduce the statement P and the statement Q." This is second part of it, applied to the first line. The fourth line introduces the hypothesis $(\neg q)$. The fifth line uses the fourth rule: "The expressions $[\neg(\neg P)]$ and P are freely interchangeable. In other words, anywhere an expression of one of these forms appears in a statement we may deduce the statement where it has been replaced by the other." This is applied to the second line. The sixth line discharges the hypothesis $(\neg q)$ from the fourth line. The seventh line uses the fifth rule of inference: "The expressions $(P \supset Q)$ and $[(\neg Q) \supset (\neg P)]$ are freely interchangeable.'' Here P is the expression $(\neg p)$ and Q is the expression q. The eighth line uses the third rule of inference:"From P and $(P \supset Q)$ we can deduce Q." This is the rule known as modus ponens. It's applied to the third and seventh lines. The ninth and final line discharges the hypothesis $[p \wedge (\neg p)]$.

The theorem $\{[p \wedge (\neg p)] \supset q\}$ is known as the "Principle of Explosion". Unlike other fanciful names, like the "Rule of Fantasy", this name is quite standard and people should understand what you're talking about if you use it. This theorem shows that from a contradiction it's possible to derive anything at all. In a theory with contradictions all statements are theorems.

A useful substitution instance of $\{[p \wedge (\neg p)] \supset q\}$ is $\{[p \wedge (\neg p)] \supset p\}$.

Here are the proofs of some other useful theorems. It would be a good idea to go through at least some of them and check that you can identify which

rule of inference is being used at each step.

1 . $(\neg p)$
2 . $[(\neg p) \wedge (\neg p)]$
3 . $[\neg(p \vee p)]$
4 . $\{\neg[(\neg p) \supset p]\}$
5 $[(\neg p) \supset \{\neg[(\neg p) \supset p]\}]$
6 $\{[(\neg p) \supset p] \supset p\}$

1 . q
2 . $[(\neg p) \vee q]$
3 . $(p \supset q)$
4 $[q \supset (p \supset q)]$
5 $\{[\neg(p \supset q)] \supset (\neg q)]\}$

1 . $(\neg p)$
2 . $[(\neg p) \vee q]$
3 . $(p \supset q)$
4 $[(\neg p) \supset (p \supset q)]$
5 $\{[\neg(p \supset q)] \supset [\neg(\neg p)]\}$
6 $\{[\neg(p \supset q)] \supset p]\}$

1 . $(p \supset q)$
2 . $[(\neg q) \supset (\neg p)]$
3 . $[q \vee (\neg p)]$
4 $\{(p \supset q) \supset [q \vee (\neg p)]\}$

1 . $[(p \supset r) \wedge (q \supset r)]$
2 . $(p \supset r)$
3 . $[(\neg r) \supset (\neg p)]$
4 . $(q \supset r)$
5 . $[(\neg r) \supset (\neg q)]$
6 . . $(\neg r)$
7 . . $(\neg p)$
8 . . $(\neg q)$
9 . . $[(\neg p) \wedge (\neg q)]$
10 . . $[\neg(p \vee q)]$
11 . $\{(\neg r) \supset [\neg(p \vee q)]\}$
12 . $[(p \vee q) \supset r]$
13 $\{[(p \supset r) \wedge (q \supset r)] \supset [(p \vee q) \supset r]\}$

As a final example we consider the tautology $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ we encountered earlier. A proof is

1	.	$[(p \supset q) \wedge (q \supset r)]$
2	. .	p
3	. .	$(p \supset q)$
4	. .	$(q \supset r)$
5	. .	q
6	. .	r
7	.	$(p \supset r)$
8		$\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$

Although formal proofs may look baffling when you first look at them it's often easy to translate them into informal proofs of a familiar type. For example, the proof above can be translated as follows.

Suppose $[(p \supset q) \wedge (q \supset r)]$ is true. Then $(p \supset q)$ and $(q \supset r)$ are both true. If p is true then q is true and therefore r is true. So $(p \supset r)$. This holds under our assumption that $[(p \supset q) \wedge (q \supset r)]$ is true, so $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$.

Substitution

This is where we should talk about the rule of substitution. A very common style of mathematical argument, and one which we formalised in the tableau method, is case by case analysis. The simplest type of case by case analysis is where we have only two cases, one where a certain statement is true and one where it's false. If we can prove a certain conclusion in both of those cases then that conclusion must be true. Or at least it must be if you accept the law of the excluded middle. Intuitionists don't.

But to apply the case by case analysis above we need the law of the excluded middle for expressions and not just for variables. In other words we need $[P \vee (\neg P)]$ to be a theorem for every expression P. There are three ways of accomplishing this. One is to run exactly the same argument as above with p replaced everywhere by P. The rules we used, of which there were only two, refer to expressions rather to variables and so no change is needed. We could do the same with $\{p \supset [q \supset (p \wedge q)]\}$, by the way. For any expressions P and Q we could replace every p by the expression P and every

q by the expression Q in the proof of $\{p \supset [q \supset (p \wedge q)]\}$ and obtain a proof of $\{P \supset [Q \supset (P \wedge Q)]\}$. A disadvantage of this approach is that we need to repeat the argument for each P and Q we need to result for. We can't just do it with the letters P and Q in place of the letters p and q because P and Q aren't even elements of our language, just elements of the language we use to talk about our language. We have to substitute the actual expressions, and so we'll need to redo that work whenever we need the result for a different pair of expressions.

Another option is to leave the realm of formal proof and enter the realm of semiformal proof. The argument above shows that for any expressions P and Q the statements $[P \vee (\neg P)]$ and $\{P \supset [Q \supset (P \wedge Q)]\}$ are theorems. Anything you can derive from them using our rules of inference is also a theorem. In other words, substitution is a derived rule of inference for this system, just as it was for the Nicod system. But now we're not giving proofs of statements but rather proofs that statements have proofs. That's a semiformal proof rather than a formal one.

The third option is to bring in the rule of substitution, which was a rule of inference in the Łukasiewicz system. This seems redundant, since we've just seen how one can get around the lack of a rule of substitution by just substituting expressions for variables within a proof, but it's convenient to have and we already decided we aren't trying for a minimal system.

There's a subtle danger here though. Consider the following proof:

$$\begin{array}{lll} 1 & . & p \\ 2 & . & q \\ 3 & & (p \supset q) \end{array}$$

The first step is to introduce the hypothesis p. The second is to apply the rule of substitution, replacing the variable p with the expression q. Variables are expressions so this is okay. The third step is to discharge the hypothesis. But $(p \supset q)$ is not a tautology. There is an interpretation under which it is false, namely the one where p is assigned the value true and q is assigned the value false, and therefore it should not be a theorem.

What went wrong? There's a difference in interpretation between statements in the Nicod or Łukasiewicz systems and in a natural deduction system. In the Nicod or Łukasiewicz systems every statement is meant

to be unconditionally true. We could stop a proof at any point and have a proof of the last statement before we stopped. This is not the case for natural deduction systems. Only statements in global scope are meant to be unconditionally true. All other statements are meant to be true if all the active hypotheses for their scope are true. In other words they're only conditionally true. That's why I had to specify that only statements in global scope are theorems. The problem with the argument above is that once we've introduced p as a hypothesis it's no longer just any variable. It's the specific variable whose truth everything will be dependent upon until we discharge that hypothesis. Replacing it with some other variable, or some other expression, can't safely be allowed.

How can we repair this? We could sacrifice the rule of fantasy but the rule of fantasy is the foundation of our system. It is literally impossible to prove anything without it. We could limit our rule of substitution by saying that only statements with global scope are available for substitution. This is a sound rule of inference. We know it's sound because the other rules are sound and this one is redundant. Anything we could prove with it we could also prove without it, by the technique we discussed earlier of repeating the proof, but with expressions substituted in for the variables. It's unnecessarily restrictive though, since it can sometimes be safe to substitute for some variables in a statement within the scope of a hypothesis. The precise rule is that in any available statement we may substitute any expression for any variable which does not appear in any hypothesis which was active in its scope. In the global scope no hypotheses are active and so we can substitute for any variable, but elsewhere certain variables will not be substitutable. Note that which variables are substitutable depends on the scope of the statement into which we're substituting, not on our current scope at the point where we want to make the substitution.

This is the first example we've seen of a phenomenon where not all variables are equally variable. Some have special status in a particular context which restricts what we can do with them. It won't be the last such example. Something similar will happen when we move on to first order logic.

Although I've put in some effort above to ensure that you can substitute into statements within the scope of a hypothesis in those cases where it's safe you shouldn't generally structure your proofs in a way which makes that necessary. If you're going to need multiple substitution instances of a

statement then you should prove that statement in global scope so that it's available everywhere. Often that means writing your proofs out of order. Once you release you'll need multiple instances of a statement you need to go back and insert a proof of that statement at the start of your argument, before introducing any hypotheses which are not needed in its proof.

When reading proofs it's useful to know that people do this. If an author starts by proving a bunch of random facts whose usefulness isn't immediately apparent and which don't reappear for several pages that's not necessarily just bad exposition. It may well be that their being proved there to make it clear that they're in global scope, i.e. that their truth is not contingent on hypotheses which will be made later.

From tableaux to proofs

We can generate a formal proof in our natural deduction system from a tableau. This generally won't give a particularly efficient proof, but it will be a formal proof.

There's a preamble which we can use for all tableaux, consisting of proofs of theorems we've already proved or easy consequences of those, whose substitution instances we'll find useful. This preamble is

1 . $[p \wedge (\neg p)]$
2 . p
3 . $(\neg p)$
4 . . $(\neg q)$
5 . . $[\neg(\neg p)]$
6 . $\{(\neg q) \supset [\neg(\neg p)]\}$
7 . $[(\neg p) \supset q]$
8 . q
9 $\{[p \wedge (\neg p)] \supset q\}$

10 . $(\neg p)$
11 . $[(\neg p) \vee q]$
12 . $(p \supset q)$
13 $[(\neg p) \supset (p \supset q)]$
14 $\{[\neg(p \supset q)] \supset [\neg(\neg p)]\}$
15 $\{[\neg(p \supset q)] \supset p]\}$

16 . q
17 . $[(\neg p) \vee q]$
18 . $(p \supset q)$
19 $[q \supset (p \supset q)]$
20 $\{[\neg(p \supset q)] \supset (\neg q)]\}$

21 . $[(p \supset r) \wedge (q \supset r)]$
22 . $(p \supset r)$
23 . $[(\neg r) \supset (\neg p)]$
24 . $(q \supset r)$
25 . $[(\neg r) \supset (\neg q)]$
26 . . $(\neg r)$
27 . . $(\neg p)$
28 . . $(\neg q)$
29 . . $[(\neg p) \wedge (\neg q)]$
30 . . $[\neg(p \vee q)]$
31 . $\{(\neg r) \supset [\neg(p \vee q)]\}$
32 . $[(p \vee q) \supset r]$
33 $\{[(p \supset r) \wedge (q \supset r)] \supset [(p \vee q) \supset r]\}$

34 $(\{[(\neg p) \supset r] \wedge (q \supset r)\} \supset \{[(\neg p) \vee q] \supset r\})$
35 $\{[(\neg p) \supset r] \wedge (q \supset r)\} \supset [(p \supset q) \supset r]$

36 $(\{[(\neg p) \supset r] \wedge [(\neg q) \supset r]\} \supset \{[(\neg p) \vee (\neg q)] \supset r\})$
37 $(\{[(\neg p) \supset r] \wedge [(\neg q) \supset r]\} \supset \{[\neg(p \wedge q)] \supset r\})$

38 . $\{(\neg p) \supset [q \wedge (\neg q)]\}$
39 . $\{[q \wedge (\neg q)] \supset p\}$
40 . $(\{(\neg p) \supset [q \wedge (\neg q)]\} \wedge \{[q \wedge (\neg q)] \supset p\})$
41 . $\{(\{(\neg p) \supset [q \wedge (\neg q)]\} \wedge \{[q \wedge (\neg q)] \supset p\}) \supset [(\neg p) \supset p]\}$
42 . $[(\neg p) \supset p]$
43 . $\{[(\neg p) \supset p] \supset p\}$
44 . p
45 $(\{(\neg p) \supset [q \wedge (\neg q)]\} \supset p)$

The point of the preamble is to have lines 9, 14, 20, 33, 35, 37, and 45 available for appropriate substitutions. Most of these are previously used examples.

After copying the preamble our strategy will be to work our way through the tableau statement by statement, with each statement in tableau eventually appearing, either as is, in the case of statements to the left of the vertical line, or negated, in the case of statements to the right of the line. To start things off we introduce the negation of the statement we want to prove as a hypothesis. If we ignore branches for the moment then everything is fairly straightforward. Every statement in the tableau is derived from an earlier one using a tableau rule, of which there is one for each of the Boolean operators. When it's $\neg$ either there's nothing to do or we use our double negation rule to peel off two $\neg$'s. When the operator is $\wedge$ we can use our splitting rule. A slightly more complicated case is $\vee$. This must be on the right of the vertical line, since we're still ignoring the branching cases. In this case we use De Morgan's laws to move the $\neg$ inside the $\vee$, making it into an $\wedge$, and then use splitting. The most complicated case is $\supset$. For that we use a substitution instance of line 15 or 20, and then use modus ponens.

Branches are somewhat trickier. Whenever we branch we introduce the two possibilities as new hypotheses in turn, i.e. discharging the first one before we introducing the second. We start from a tableau proof, meaning a closed tableau, so we will eventually reach a contradiction in each branch. This contraction will be a pair of statements of the form P and $(\neg P)$ for some expression P. Once this happens we join the statements to get $[P \wedge (\neg P)]$ and then discharge the hypothesis. After discharging the second hypothesis, i.e. the one from the right branch, what we do next de-

pends on whether there are further undischarged hypotheses. If there are then we a substitution instance of one of lines 33, 35, or 37, depending on whether the operator which caused the branching was an $\vee$, $\supset$, or $\wedge$ to get a contradiction in our current scope. Once we are in the global scope, having discharged all hypotheses, we can use a substitution instance of line 45 and modus ponens to get the statement we were trying to prove, which is now in global scope and is not negated so we have a formal proof in the natural deduction system.

I'll illustrate this with the tableau for $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$, which we saw earlier.

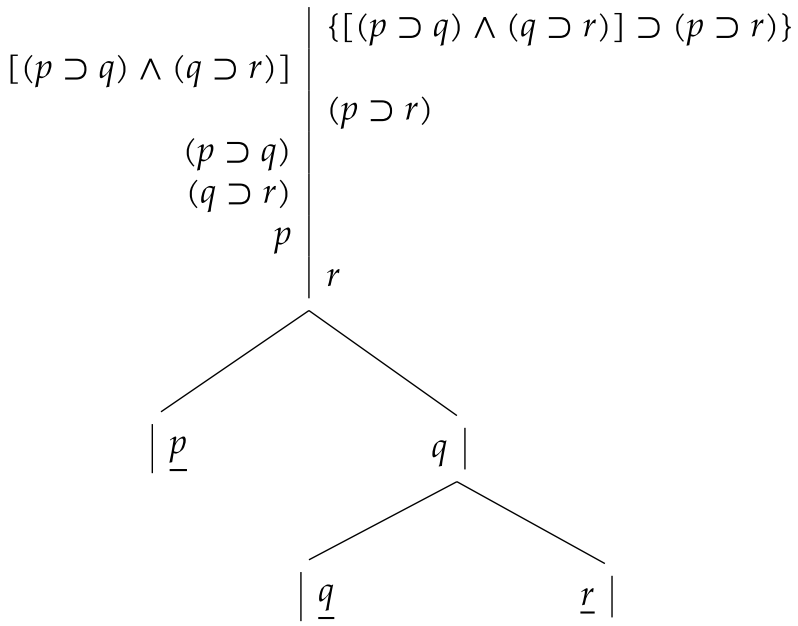


Figure 27: An analytic tableau for $[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)$

As outlined above, we start by copying the preamble and then introducing the negation of the statement we want to prove as a hypothesis.

46 . $(\neg\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\})$

This is then followed by a substitution instance of line 15, $\{[\neg(p \supset q)] \supset p]\}$, specifically the one where we substitute $[(p \supset r) \wedge (q \supset r)]$ for p and

$[(p \vee q) \supset r]$ for q, and then modus ponens

47 . $\{(\neg\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}) \supset [(p \supset q) \wedge (q \supset r)]\}$
48 . $[(p \supset q) \wedge (q \supset r)]$

We can then do essentially the same thing, but with line 20 rather than line 15,

49 . $\{(\neg\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}) \supset [\neg(p \supset r)]\}$
50 . $[\neg(p \supset r)]$

We can then apply splitting, twice, to line 48,

51 . $(p \supset q)$
52 . $(q \supset r)$

If you look at lines 46, 48, 50, 51 and 52 you'll see that they match the first five lines of the tableau, except that statements to the right of the vertical line are negated. As explained above, this is our strategy, to follow the tableau, using our rules of inference, and sometimes substitution instances of statements from the preamble, to derive each statement from earlier statements. We continue by deriving p and $\neg r$, which will take us up to the first branch point.

53 . $\{[\neg(p \supset r)] \supset p\}$
54 . p
55 . $\{[\neg(p \supset r)] \supset (\neg r)\}$
56 . $(\neg r)$

As described earlier, the strategy for dealing with branches is to introduce a hypothesis for each branch in turn and derive a contradiction under that hypothesis. We start the left branch by introducing the hypothesis $(\neg p)$

57 . . $(\neg p)$

At this point we have both p and $(\neg p)$ available within the current scope, and can join them, and use a substitution instance of line 9, before discharging the hypothesis.

58 . . $[p \wedge (\neg p)]$
59 . . $\{[p \wedge (\neg p)] \supset [s \wedge (\neg s)]\}$
60 . . $[s \wedge (\neg s)]$
61 . $\{(\neg p) \supset [s \wedge (\neg s)]\}$

Having dealt with the left branch we now proceed to the right branch. Again we introduce a hypothesis.

62 . . q

In the tableau there's another branch at this point, which we deal with in the same way as the previous one. The first step is to introduce a further hypothesis.

63 . . . $(\neg q)$

Within the current scope we have a q and a $(\neg q)$, from the previous two lines of the proof, which we can treat in the same way we treated p and $(\neg p)$ earlier.

64 . . . $[q \wedge (\neg q)]$
65 . . . $\{[q \wedge (\neg q)] \supset [s \wedge (\neg s)]\}$
66 . . . $[s \wedge (\neg s)]$
67 . . $\{(\neg q) \supset [s \wedge (\neg s)]\}$

That was the left subbranch of the right branch of the tableau. The right subbranch is handled similarly.

68 . . . r
69 . . . $[r \wedge (\neg r)]$
70 . . . $\{[r \wedge (\neg r)] \supset [s \wedge (\neg s)]\}$
71 . . . $[s \wedge (\neg s)]$
72 . . $\{r \supset [s \wedge (\neg s)]\}$

Now we need to combine the two subbranches. We do this by joining lines 67 and 72, substituting in line 35, applying modus ponens twice, and then discharging the hypothesis q.

$$
\begin{array}{lll}
73 & . \ . & (\{(\neg q) \supset [s \wedge (\neg s)]\} \wedge \{r \supset [s \wedge (\neg s)]\}) \\
74 & . \ . & [(\{(\neg q) \supset [s \wedge (\neg s)]\} \wedge \{r \supset [s \wedge (\neg s)]\}) \\
 & . & \quad \supset \{[\neg(q \supset r)] \supset [s \wedge (\neg s)]\}] \\
75 & . \ . & \{[(q \supset r)] \supset [s \wedge (\neg s)]\}] \\
76 & . \ . & [s \wedge (\neg s)]\} \\
77 & . & \{q \supset [s \wedge (\neg s)]\}
\end{array}
$$

Then we combine the two branches in a similar way.

$$
\begin{array}{lll}
78 & . & (\{(\neg p) \supset [s \wedge (\neg s)]\} \wedge \{q \supset [s \wedge (\neg s)]\}) \\
79 & . & [(\{(\neg p) \supset [s \wedge (\neg s)]\} \wedge \{q \supset [s \wedge (\neg s)]\}) \supset \{[\neg(q \supset r)] \\
 & & \quad \supset [s \wedge (\neg s)]\}] \\
80 & . & \{[(p \supset q)] \supset [s \wedge (\neg s)]\}] \\
81 & . & [s \wedge (\neg s)]\} \\
82 & & \{(\neg\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}) \supset [s \wedge (\neg s)]\}
\end{array}
$$

Now we apply a substitution instance of line 45 and then modus ponens to complete the proof.

$$
\begin{array}{ll}
83 & (\{(\neg\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}) \supset [s \wedge (\neg s)]\} \\
 & \quad \supset \{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}) \\
84 & \{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}
\end{array}
$$

After 84 lines we've finally proved $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$. We proved it earlier in 8 lines, so why give a proof which is more than ten times longer? The point is that this proof was constructed in a purely mechanical way from the tableau. We can use the same method on any closed tableau to get a formal proof. The preamble will always be the same, as will the strategies for dealing with branches and contradictions, and the use of line 9 to complete the proof. This means tableaux can now be regarded as semiformal proofs rather than informal proofs, since a tableau tells us not just that a statement is true but also that it's a theorem in our formal system.

Soundness, consistency and completeness

Soundness of a formal system means that the axioms are true in every intended interpretation and that no rule of inference can derive a false statement from true ones in any of the intended interpretations. From this it follows that every theorem is true in every intended interpretation. Consistency means that a statement and its negation cannot both be theorems. Consistency doesn't depend on the choice of interpretations or interpretations, except to the extent that we need to identify what negation means in the system. Completeness means that every statement which is true in every intended interpretation is a theorem.

The intended interpretations of our natural deduction system are the ones discussed earlier for zeroeth order logic in general. There is an interpretation for each possible assignment of truth values to variables.

Our natural deduction system is sound. There are no axioms so it's trivially the case that every axiom is true in every intended interpretation. It's also impossible to derive a false statement from true ones in any of the intended interpretations.

If you believe that a statement and its negation can't simultaneously be true then consistency follows from soundness. It's also possible to prove the consistency of natural deduction directly, but I won't.

The completeness of the natural deduction system follows from two facts proved earlier, that every tautology has a closed tableau and that any closed tableau can be converted into a formal proof.

For the two axiomatic systems considered earlier it's fairly easy to prove soundness. For Łukasiewicz it's also straightforward to prove consistency. For Nicod the main obstacle to proving consistency is the lack of a negation operator in the language. Proving completeness of either system is possible, but is more difficult than for natural deduction.

Non-deterministic algorithms

It's often useful to think in terms of non-deterministic calculations. In some sense, as we'll see, any non-deterministic computation can be turned into a deterministic one, but often the non-deterministic formulation is more

natural. By non-deterministic I do not mean probabilistic. Probabilistic computation is also often useful, but is not really relevant to this module.

Deterministic algorithms have

- a well defined initial state
- a definite rule for what action to take in each state, possibly dependent on input, environment, etc.
- a termination condition
- (probably) a distinction between successful and unsuccessful termination

Non-deterministic algorithms have

- one or more initial states
- a set of possible actions to take in each state
- a termination condition, probably with a distinction between successful and unsuccessful termination

Usually we're interested in whether a non-deterministic algorithm can terminate successfully, not whether it happens to for a particular choice of actions, so unsuccessful terminations are generally to be regarded as unsuccessful only in a local sense. There may be other computational paths which lead to successful termination.

We usually illustrate the possible computational paths with a tree diagram. Unfortunately the terminology for trees is a mess. In keeping with the tree metaphor we have the "root" and "leaves". For some reason trees are usually drawn growing downwards, so the "root" is at the top of the diagram. We also have "parents" and "children", though. Every node has exactly one parent, except the root, which has none. Leaves are nodes with no children. Nodes which share the same parent are called "siblings". Similarly we talk about "ancestors" and "descendents", which mean what you would guess, based on the family tree metaphor, except that usually we consider each node to be one of its own ancestors and descendents.

Puzzles

Many combinatorial puzzles are easily formulated as non-deterministic computations. Consider, for example, the classical eight queens problem. The start state is an empty board. The possible actions at any point are placing a queen in any of the squares not accessible in a single move from those already on the board. If there are none then the computation terminates, successfully, if we've placed eight queens, and unsuccessfully, if we haven't.

The problem can be made more tractable if we realise that any successful solution must have exactly one queen in each row and the order in which we place the queens has no bearing on whether the configuration is permitted or not, so we can reformulate the problem slightly, with the permitted actions at the i's step being placing a queen somewhere in the i'th row which is not accessible in a single move from those already on the board. This change, for example, reduces the number of possible actions for the first step from 64 to 8 and the number for the second step from somewhere between 36 and 42 to either 5 or 6. The number of possibilities for the full problem is still too many for diagrams which fit on a single page though so we'll consider a chessboard with four rows and columns rather than eight.

A good way to illustrate the possible computational paths is with a tree, as in the diagram.

The lowermost two leaves of the tree, the ones with four queens placed, represent successful computational paths. The other four leaves, with either two or three queens placed, represent unsuccessful computational paths.

We can, of course, do the same thing in the original eight queens problem or, more generally, in a wide variety of similar puzzles.

Linguistic examples

Generating elements of a language can be considered as a non-deterministic computation. Specifically, we can treat the problem of recognising whether a given list of tokens is an element of the language in the following way.

- States correspond to lists of symbols or tokens.

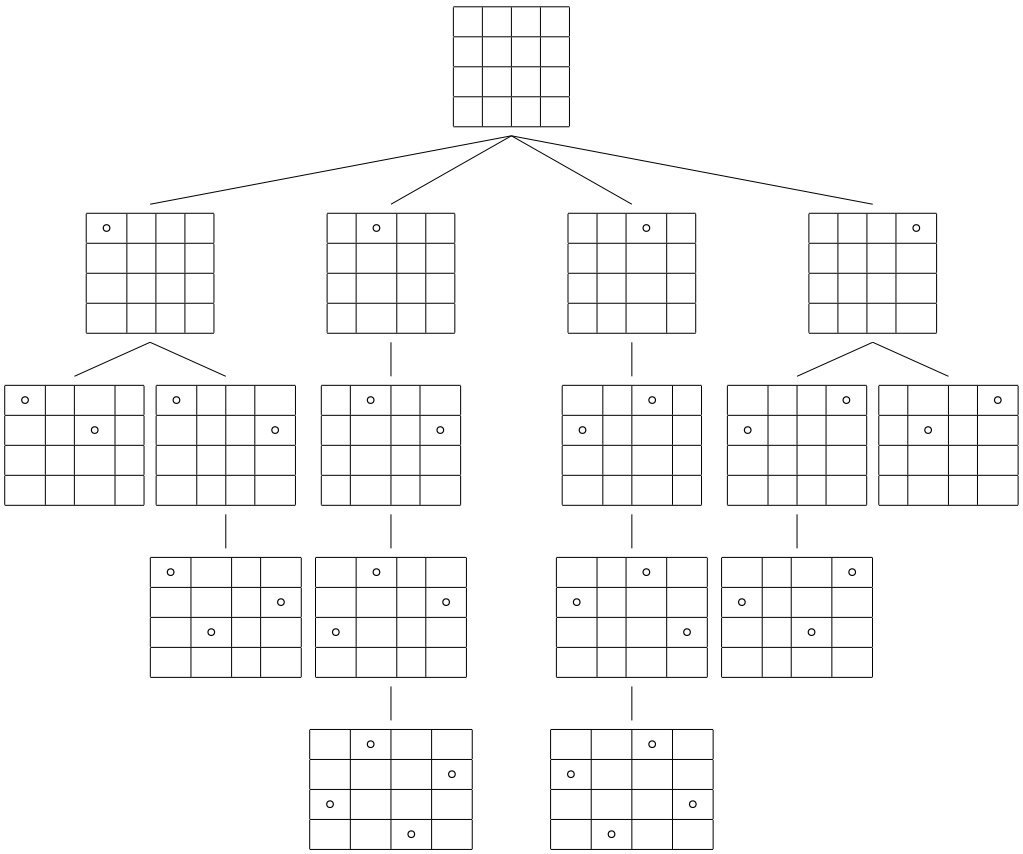


Figure 28: Solution of the four queens problem

- The initial state is a list with just the start symbol.
- The allowed actions are replacing a terminal symbol from the list by a token, or a non-terminal by one of its alternates.
- Successful termination happens when the list matches the input.
- Unsuccessful termination is when the list consists only of tokens, and so no further actions are possible, but the list does not match the input.

As usual, an unsuccessful termination just means that the particular choices of actions in this computational path did not succeed. Some other set of choices might.

As with the puzzle example, it's possible, and useful, to restrict the allowed actions in such a way that if the original problem is solvable then so is the restricted one. In this case we can do this by insisting that expansion is only applied to the first symbol in the list. Using the last would work equally well, but we'll use the first.

As an example, consider the following grammar for the language of balanced parentheses.

```
ok : | "(" ok ")" ok
```

and the input "(()())". This has balanced parentheses and so should be recognised. The full tree is infinite, so we'll have to examine it in parts.

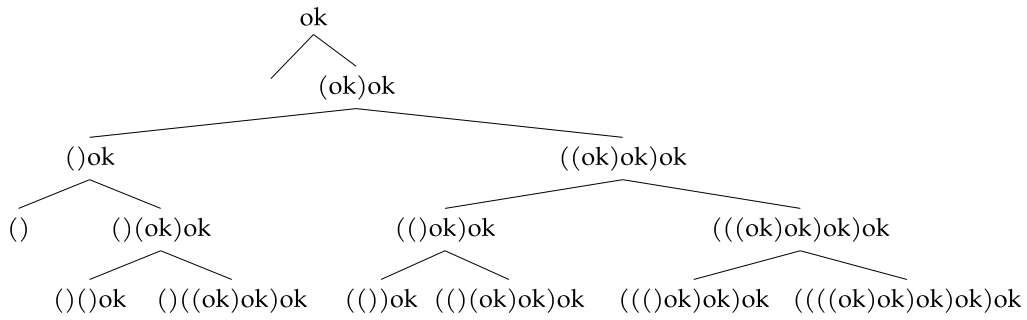


Figure 29: Recognising (()()), part 1

The first figure shows the possible states after four steps, and the computational paths which lead to each one. In this case there are only two possible

expansions of ok, the only nonterminal symbol, so we have a binary tree, i.e. one where each node has at most two children. There are two cases where there are no children, the strings "" and "()", which represent unsuccessful terminations, since neither of them is our input string. We could continue, but the tree is growing exponentially and some pruning is advisable. We can notice, for example, that proceeding from the root and following the right branch and then the left leads us to "()ok". This computation is doomed. No matter what choices we make from this point on we can only get strings starting with "()", which can never match the given input "(()())". So we can ignore that branch. We can, of course, also ignore the original left branch, which terminated unsuccessfully immediately. So the only computational paths which are potentially relevant are those which start by going right, and then right again. If we go right one more time though then we reach "(((ok)ok)ok)ok". Any possible continuation from here will lead to a string starting with "(((", which also cannot match our input. So we need only consider the part of the tree where we choose right, right and left initially, leading to "(()ok)ok)". The next diagram shows the part of the tree starting from that node.

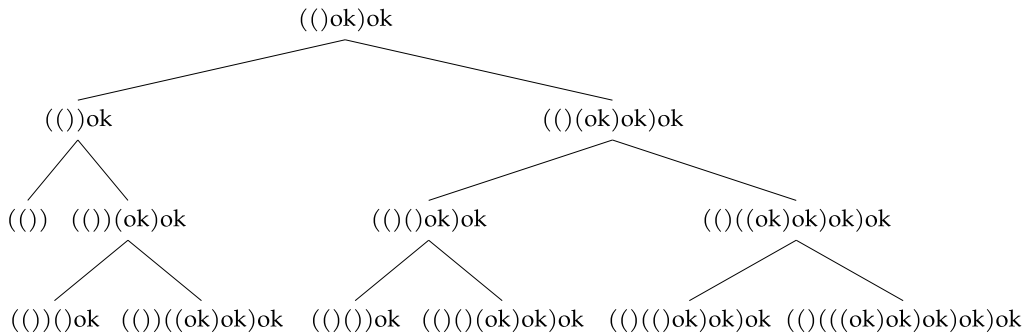


Figure 30: Recognising (()()), part 2

In this new diagram we see another unsuccessful termination, for the branch ending in "(())". If fact the whole left branch is doomed, since it can only generate strings starting with "(())", as is the the right subbranch of the right branch, since it can only generate strings starting with "(()((". We can therefore restrict our attention to the part of the tree from the left subbranch of the right branch, starting from the state "(()()ok)ok", which is shown in the next diagram.

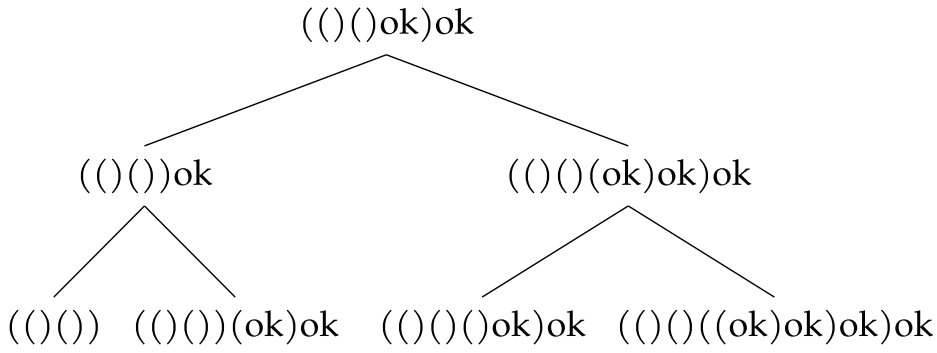


Figure 31: Recognising (()()), part 3

On the far left of that diagram we see the string "(()())", which matches our input string, so this computational path succeeds and the string is recognised.

A single successful computational path is enough so we don't need to worry about what would happen to any of the computations which have not terminated, although it turns out none of them would be successful.

I've described this as a recogniser, but it's not too difficult to turn it into a parser. The only change which is needed is to keep track of the computational path which led us to each state, since this contains the information about how each symbol was expanded, which is all we need in order to construct a parse tree.

What would have happened if the input string was invalid, i.e. did not have balanced parentheses? This depends on whether we prune the tree systematically as in the example above. If we do then eventually all computational paths will terminate unsuccessfully. If we don't then the computation will simply run forever. For valid inputs pruning makes the algorithm more efficient, and certainly easier to produce diagrams for, but isn't essential for it to recognise the input as valid.

In our example the tree was a binary tree. This happened for two reasons. First, all of our non-terminal symbols, of which in fact there was only one, had more than two alternates. Second, none of our nonterminal symbols, which were "(" and ")" had more than one token belonging to it. In fact they each had only one, but we would still have got a binary tree even if

they'd had two.

If we consider other grammars the number of alternates for a non-terminal may be larger than two, but it will always be finite. The number of tokens belonging to a symbol could be infinite though, in which case we will have nodes with infinitely many children. We'll see how to deal with this later.

Zeroeth order logic as a non-deterministic computation.

The satisfiability version of the tableau method for zeroeth order logic can be viewed as a non-deterministic algorithm.

- The initial state is one with the statement to the left of the line.
- At each step the actions are to take an unused statement, mark it used, and write down its consequences, in the non-branching case, or one of the two possibilities, in the branching case.
- The termination condition is running out of unused statements (successful) or finding a contradiction (unsuccessful).

If the algorithm can terminate successfully then the statement is satisfiable.

In this case, unlike many non-deterministic computations, the algorithm will always terminate, successfully or not.

Similar remarks apply to proving tautologies, but we start with the statement to the right of the line and "success" and "failure" mean the opposite of what you'd expect.

The Łukasiewicz formal system or, more generally, any axiomatic system can also be considered a non-deterministic algorithm.

- The initial state is an empty list of statements.
- In each state our options are to write down an axiom or apply a rule of inference to one or two previous statements.
- Successful termination is writing down the statement we were trying to prove.
- Unsuccessful termination is impossible, but non-termination is very common.

The given statement is a theorem if the algorithm can terminate successfully.

One complication is that substitution allows replacement of a variable by any expression and there are infinitely many expressions, so again we have to consider trees where some nodes may have infinitely many children.

Trees of trees

In some cases, like the parsing method described above or the algorithm for constructing tableaux, we have a non-deterministic algorithm for constructing a tree. There is then a tree diagram for the computational paths, where each node is a possible state of the system, which is generally are partially filled in tree of the type the algorithm is meant to construct. In other words we have a tree of trees. The trees associated to the nodes are all finite trees, because they are the result of a finite computation, but the larger tree, representing all possible computational paths, is generally infinite.

Sometimes different program paths lead to identical states. You can take advantage of this by replacing the state tree with a "directed acyclic graph", so instead of a tree of trees we have a directed acyclic graph of trees. There are practical parsing algorithms which use this trick to avoid exponential growth.

Making non-deterministic algorithms deterministic

How do we know whether the algorithm can terminate successfully? One idea is to simulate all possible choices, so if one of them works we won't miss it. This is more or less what we did for zeroeth order logic. The tree structure of the tableau keeps track of the possible choices for us. We can apply the same trick essentially any non-deterministic algorithm, using a tree to keep track of the choices. There are some complications though if there are sometimes infinitely many choices.

To check whether the algorithm could successfully terminate we have to visit every node in the tree, but there might be infinitely many. There are two standard ways to traverse a tree, depth first or breadth first. Depth first means visiting all of a node's children before moving on to any of its

siblings. Breadth first means visiting all of a node's siblings before moving on to any of its children. The diagrams show both a depth first and a breadth first traversal of a binary tree with seven nodes, numbered in the order in which they are traversed. Depth first is usually easier to program, so if you've used a program which traverses a tree that's probably what it did.

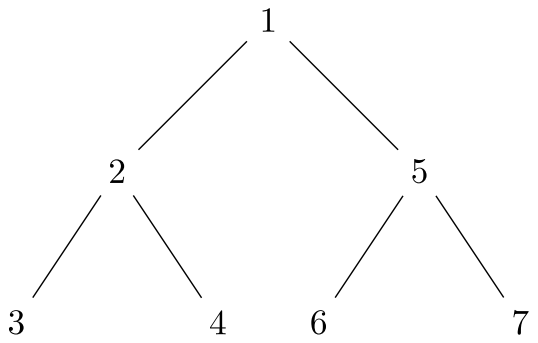


Figure 32: depth first tree traversal

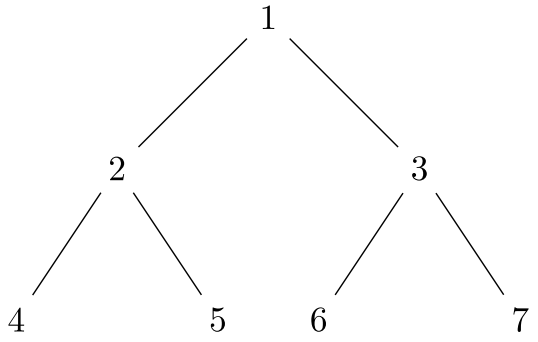


Figure 33: breadth first tree traversal

Depth first traversal will work, i.e. visit every node, on finite trees and some, but not all, infinite trees. Breadth first traversal will work, even on infinite trees, provided no node has infinitely many children. "Working" means that if what we're looking for is there we will find it. If it's not there the traversal will continue forever, unless the tree happened to be finite.

The trees in the tableau method for zeroeth order logic were finite, so either method works. The trees for Łukasiewicz have branches of finite length,

but infinitely many children. Neither method works for this. The trees (of trees) for parsing usually have infinitely long branches but have only finitely many children, unless there are symbols with infinitely many tokens, so breadth first generally works and depth first generally doesn't.

There are traversal methods which work even for trees with infinite branches and infinitely many children, provided the infinities aren't too bad. When there are more than two children we can group them as the first one and the others, then create a new node for the others. The result is a binary tree containing all the nodes of the original tree, on which we can do breadth first traversal. This is illustrated in the two accompanying diagrams, but infinite children are hard to draw so we'll have to make do with three. The first shows a non-binary tree and second shows the corresponding binary tree, with extra nodes added.

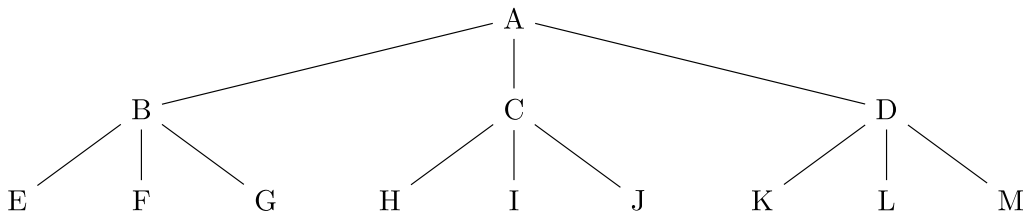


Figure 34: a non-binary tree

First order logic

The next step after zeroeth order logic is first order logic. In fact it's also normally the last step. Higher order logic exists as well, but the standard formulation of mathematics makes no use of it, using only first or logic and set theory as its foundations. We'll talk about set theory later and first order logic now.

The most important thing which first order logic introduces is quantifiers, specifically the universal quantifier "for all" and the existential quantifier "for some".

There are some other new elements as well. One is "predicates". The term is unfortunate. It's borrowed from linguistics, but in a way which is incom-

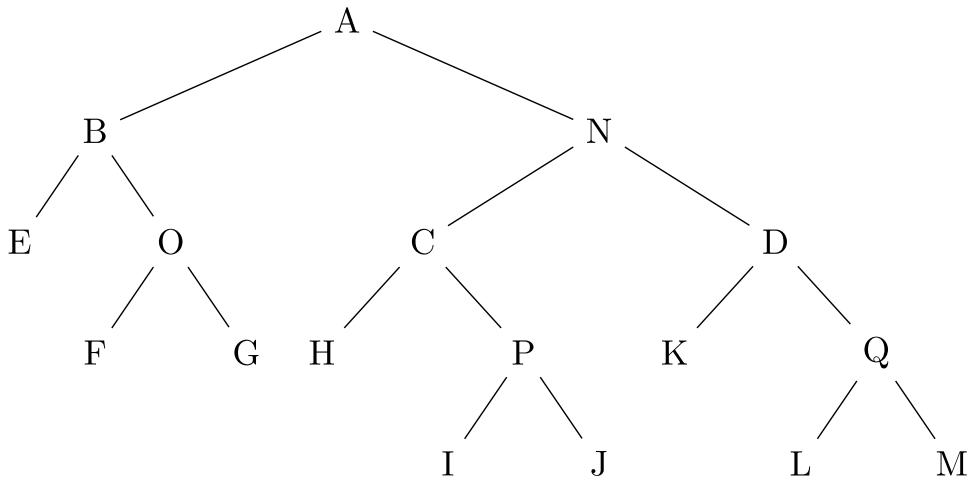


Figure 35: the corresponding binary tree

patible with the way it's used there. A word which better reflects the role they play might have been "property", but "predicate" is standard.

First order logic can be difficult to follow though if you have no examples in mind. We'll use first order logic soon in discussing the integers and sets, so it may be helpful to give some examples chosen from those.

For integer arithmetic "... is prime" is an example of a unary predicate, i.e. one which takes a single variable. The variable in this case is an integer and the value of the predicate is true or false depending on whether that integer is or is not prime. One the other hand "... is less than ..." is a binary predicate, one which takes two values and is either true or false depending on whether the first is less than the second or not. There are also ternary predicates, with three variable, like "... is the sum of ... and ...", which is true if first variable is the sum of the second and third. For set theory an important unary predicate is "... is finite". A binary predicate is "... is a member of ...". A ternary predicate is "... is the union of ... and ...".

Although the examples above may be helpful in understanding what role predicates play first order logic doesn't concern itself with the meaning of predicates and indeed has no way to represent any such meaning. Predicates appear as letters, just as variables do. This letter is just as unspecified in its meaning as a variable is. First order logic doesn't care whether a

ternary predicate represents "... is the sum of ... and ..." or "... is the union of ... and ..." or something else entirely.

In addition to quantifiers, variables and predicates we also have "parameters". It's somewhat harder to describe what a parameter is. When we discussed the rule of substitution in zeroeth order logic we saw that not all variables are equally variable. Some are allowed to vary more than others. Parameters represent instantiated variables. Suppose, for example, we're operating in a context where we have available the statement that for every integer n there is a prime number greater than n. This statement, which has a universal quantifier in the "for every" and an existential quantifier in the "there is", might be available because it's already been proved or it might be available because we've introduced it as a hypothesis, as we discussed when we talked about natural deduction. In this case the statement happens to be true but it could conceivably have appeared in a proof by contradiction. In any case we have the statement available. Now 2024 is an integer, so we are assured by this statement that there is a prime number greater than 2024. We could then say "let p be such a prime". In that case a logician would describe p as a parameter. Like variables, parameters are allowed to appear as arguments of predicates. Mathematicians tend not to bother with such distinctions and would refer to both n and p as variables but logicians are more careful and distinguish between variables and parameters because the rules of inference treat them somewhat differently, as we'll see.

The distinction between variables and parameters may become clearer when we apply first order logic in other theories, like elementary arithmetic or set theory. In arithmetic, for example, the variables will be numerical. We will also have numerical expressions, like $n + 3$. When applying first order logic we will be able to substitute numerical expressions for parameters, but not for variables.

We'll also retain the logical operators of zeroeth order logic along with the parentheses but we'll leave behind the Boolean variables. The things connected by the operators will be expressions built from predicates. We still have variables but they are not, or at least are not necessarily, Boolean variables and it will not make sense to talk about variables being true or false. In integer arithmetic the variables will represent numbers and in set theory they will represent sets. In neither case does it make sense to ask whether

they are true or false.

The version of first order logic we'll consider is "untyped". In other words there is only one set of variables. This isn't the way mathematicians or computer scientists are used to working. If we're discussing linear algebra we might, for example, want to have three types of variables, for scalars, vectors and matrices. Traditional linear algebra textbooks use lower case Greek letters for scalars, bold lower case Latin letters for vectors, and upper case Latin letters for matrices, for example. While that's often convenient it's not strictly necessary. We could accomplish the same thing by having a single set of variables and introducing the unary predicates "... is a scalar", "... is a vector" and "... is a matrix". In fact we'd be better off not introducing the last of these and simply thinking of vectors as matrices with only one column and of scalars as vectors with only one row. First order logic makes no assumptions about what this one type of variable might represent.

Varieties of first order logic

There's really one zeroeth order logic. We saw a few different languages, differing in precisely which Boolean operators they allowed, and different sets of axioms and rules of inference, but any statement expressible in one of these languages was also expressible in any of the others and the translated version was a theorem in the new system if and only if the original version was a theorem in the old system, although I don't claim this is obvious.

First order logic is not as not as unified as zeroeth order logic. There are concepts which appear in some varieties of first order logic but which are absent in others, and aren't even expressible in them. An example is the concept of equality. Some variants of first order logic have an "=" symbol and axioms or rules of inference governing its use. Others don't have an "=" symbol. This isn't like zeroeth order logic though, where some variants have a "$\supset$" symbol and others don't, but can express the same meaning with a combination of other symbols. First order logic without equality is simply incapable of expressing the notion of equality with any combination of symbols. If you want to use first order logic as the foundation for a subject with a notion of equality, like arithmetic, then you need to add axioms or rules of inference to deal with the "=" symbol, just as you to with,

for example, the "+" symbol. If you use first order logic with equality then you don't have to do this, since "=" already belongs to the logic, just like Boolean operators or quantifiers.

Unfortunately, even where different variants of first order logic use the same language they may differ in the interpretations they allow, and therefore in which statements are considered valid, i.e. true in all interpretations. One important point of contention is whether all parameters are assumed to refer to actually existing objects of the appropriate type. If you decide that they are, and build a system for arithmetic on top of it in the way that we will, allowing numerical expressions to be substituted for parameters in any valid statement from first order logic, then you have to be very careful to make sure that every expression your language allows you to write down will always refer to a number no matter what numerical values are assigned to its variables. Unfortunately that means you have to avoid introducing things like a division sign, since m/n won't exist for some choices of m and n.

Most logic textbooks introduce a form of logic without equality but with existential presuppositions, i.e. with the tacit assumption that every parameter refers to some existing object of the appropriate type. When I say that this assumption is tacit I mean that it's not made explicit but the axioms and rules of inference are not sound if it's violated.

Mathematicians rarely specify the underlying logic of their systems but if you look at what they do you'll see that the foundation is usually a logic with equality and without existential presuppositions. They won't introduce extra axioms or rules of inference for "=", just as they don't for Boolean operators or for quantifiers. They will introduce symbols like "/" despite the fact that this allows us to write down expressions which may not refer to any existing object.

If I were writing a text where logic was the main subject of study rather than mostly a foundation for other theories I might discuss several different variants of first order logic and explore their differences but we don't have time for that and so I'll pick one, the one closest to actual mathematical practice: first order logic with equality and without existential presuppositions. This introduces extra complications into the logic, compared to a logic without equality and with existential presuppositions, but makes using as a foundation for other theories both easier and safer. The specific

system described below is in essence due to Jaakko Hintikka.

A language for first order logic

The language has been described informally above but here is a formal grammar for it.

```
statement    : expression
expression   : atomic_expr
             | "(" expression binop expression ")"
             | "[" expression binop expression "]"
             | "{" expression binop expression "}"
             | "(" "¬" expression ")"
             | "[" "¬" expression "]"
             | "{" "¬" expression "}"
             | "(" quantifier variable "." expression ")"
             | "[" quantifier variable "." expression "]"
             | "{" quantifier variable "." expression "}"
atomic_expr  : "(" atom ")" | "[" atom "]" | "{" atom "}"
atom         : predicate | atom individual
             | individual "=" individual
individual   : variable | parameter
binop        : "∧" | "∨" | "⊃"
quantifier   : "∀" | "∃"
predicate    : pred_letter | predicate "!"
pred_letter  : "f" | "g" | "h" | "i" | "j"
parameter    : param_letter
             | parameter "!"
param_letter : "a" | "b" | "c" | "d" | "e"
variable     : var_letter
             | variable "!"
var_letter   : "v" | "w" | "x" | "y" | "z"
```

As in the zeroeth order calculus, exclamation points can be used to generate an infinite number of predicates, variables and parameters, but we will never actually need them in examples.

As with zeroeth order logic it's useful to have symbols which don't belong to the language but which are used for talking about the language. We'll

continue to use P, Q, etc. to represent expressions but we'll add A, B, etc. for parameters, F, G, etc. for predicates, and V, W, etc. for variables. The same convention about different types of brackets being interchangeable applies.

Free and bound variables

One of the most confusing, but also most important, parts of first order logic is the distinction between free and bound variables, or, more properly, between free and bound occurrences of a variable in an expression. This is easier to understand in a formal system like the one we will use for elementary arithmetic than in first order logic so we'll consider it there first.

Consider the expression

$$l = m + n.$$

This could be true or false, depending on the values of l, m and n. Now consider the expression

$$[\exists n.(l = m + n)]$$

which is normally read "there exists an n such that l equals m plus n". This could be true or false depending on the values of l and m. You could substitute actual natural numbers in for l and m and sensibly ask whether this is a true statement for those values. In the version of elementary arithmetic we'll consider the variables represent natural numbers, i.e. non-negative integers so this statement will in fact be true if $l \geq m$ and false if $l < m$. What the value of the expression doesn't depend on is n. You are not allowed to substitute in a value for n. The result of doing so isn't true or false but just grammatically incorrect. n is what's called a bound variable in this expression, while l and m are free variables. In the original expression all three variables were free.

We can add another quantifier.

$$\{\forall l.[\exists n.(l = m + n)]\}$$

is normally read "for all l there exists an n such that l equals m plus n". The value of the expression now depends only on l. In fact it's true if m is zero and is false if m is positive. In this expression l and n are bound while m is free.

We can add one final quantifier.

$$(\exists m.\{\forall l.[\exists n.(l = m + n)]\})$$

Now all the variables are bound. It would not make sense to substitute for any of them. This statement is either true or false, not depending on anything. In fact it is true, since zero is an example of such an m. In fact its the only example.

It's important to note that we can only talk about whether a variable is free or bound within a particular expression. Each of the first three expressions above forms part of the expression which follows it and there is a variable which is free in the subexpression but bound in the whole expression. More subtly the same variable could be free and bound in different places in the same expression. This doesn't happen in any of the expressions above and you should never write down such an expression because they are very confusing. I will try not to either. But it's hard to write down grammar rules which forbid this so the standard practice is to allow it but then not do it. Because of this though we have to talk about free and bounded occurrences of a variable in an expression rather than free and bound variables, since a variable could potentially occur freely in one place and bound in another within the same expression.

The example above was taken from elementary arithmetic. The corresponding example in first order logic would be the four expressions

$$(fxyz),$$

$$[\exists z.(fxyz)],$$

$$\{\forall x.[\exists z.(fxyz)]\},$$

and

$$(\exists y.\{\forall x.[\exists z.(fxyz)]\}).$$

The particular predicate saying that the first argument is the sum of the last two has been replaced by the generic symbol f. I've also renamed the variables to put them within the range specified by the grammar. Whether the last of these statements is true or false depends on the meaning of f. If f is the sum predicate we considered earlier then the statement is true but for other choices of f it might be false. This is a question of interpretation, which we'll discuss later.

The variable x has a free occurrence, but no bound occurrences, in the first and second expressions above and has a bound occurrence, but no free occurrences, in the third and fourth. y has free occurrences in the first three and a bound occurrence in the fourth. z occurs freely in the first and bound in the last three.

The precise rules are not difficult to state. In an atomic expression all variables are free. Whenever we build an expression from a quantifier, a variable, and an expression all occurrences of that variable in the combined expression are bound. Occurrences of other variables remain free or bound as they were in the original expression. Combining expressions using logical operators has no effect. Whatever occurrences were free in the old expressions remain free in the new one and whatever occurrences were bound remain bound.

An expression is called "open" if there is a free occurrence of some variable in it and is called "closed" if all occurrences are bound. If we have a particular interpretation where the variables are assumed to belong to a particular set and assigned particular relations to the predicates then it makes sense to ask whether a closed expression is true. For an open expression we can only ask that question after we've assigned particular values to the variables which occur freely in the expression.

Our grammar only allows quantifiers to appear before variables so it's not meaningful to talk about free or bound occurrences of a parameter, only of a variable.

A closed statement without parameters is called a "sentence". These are the only statements about which we can reasonably ask whether they are true or false in an absolute sense, rather than for particular values of the variables and parameters appearing in them.

Interpretations

Interpretations in zeroeth order logic were relatively simple. For each variable we got to assign it one of two values, true or false. Technically there were infinitely many variables and hence infinitely many interpretations but for any particular statement, or finite set of statements, only finitely many variables occur and so we could enumerate all the interpretations.

This was in fact the basis of the method of truth tables.

First order logic has many more interpretations. We can, for example, construct an interpretation as follows. We begin by choosing a pair of sets, called the "inner domain" and "outer domain", such that the inner domain is a subset of the outer domain. Then we assign an element of the domain to each variable and an element of the outer domain to each parameter. To each predicate we assign a relation, which you can safely think of as a Boolean-valued function, on the outer domain. To each unary relation we assign a unary relation, which you should think of as a function which takes a single argument, belonging to the domain, and gives you a Boolean value, i.e. true or false. To each binary predicate we assign a binary relation, i.e a Boolean function of two arguments. To each ternary predicate we assign a ternary relation, and so forth. An atomic expression is considered true if the function associated to its predicate, when evaluated at its arguments, has the value true. Then from the atomic expressions we can assign truth values to larger and larger expressions, much as we did in zeroeth order logic. We handle Boolean operator just as we did there. Equality is handled in the obvious way, with $(A = B)$ evaluating to true if the two variables or parameters have been assigned the same value. The one complication is that when we assign truth values to a quantified expression we check only those values of the variable in the inner domain, and we substitute them only for free occurrences of the variable in the inner expression.

A statement in first order logic is said to be valid if it is true for every interpretation of the type described above. Valid statements play much the same role for first order logic as tautologies did for zeroeth order logic.

There is no analogue of truth tables for first order logic because we have no hope of listing the possible interpretations of a statement.

Some textbooks imply, or even directly state, that all interpretations are of the type described above. They are wrong, for reasons we will discuss in the set theory chapter.

Informal proofs

We can construct informal proofs in first order logic in much the same way we did in zeroeth order logic, but asking how the given statement could be

false and trying to show that none of these ways can actually occur.

As an example, consider the statement

$$\{[\exists x.(fx)] \supset [\neg(\forall x.\{\neg[(fx) \vee (gx)]\})]\}.$$

Suppose it were false in at least one interpretation. This is of the form $(P \supset Q)$, where P is the expression $[\exists x.(fx)]$ is the expression $[\neg(\forall x.\{\neg[(fx) \vee (gx)]\})]$. We know that an expression of the form $(P \supset Q)$ can only be false when P is true and Q is false, so $[\exists x.(fx)]$ should be true and $[\neg(\forall x.\{\neg[(fx) \vee (gx)]\})]$ should be false. We also know that an expression of the form $(\neg P)$ can only be false if P is true, so $(\forall x.\{\neg[(fx) \vee (gx)]\})$ should be true. If $[\exists x.(fx)]$ is true then there is some a such that fa. If $(\forall x.\{\neg[(fx) \vee (gx)]\})$ is true then $\{\neg[(fa) \vee (ga)]\}$. Since this is true $[(fa) \vee (ga)]$ must be false. Then (fa) is also false. But we previously found (fa) to be true. So the assumption that $\{[\exists x.(fx)] \supset [\neg(\forall x.\{\neg[(fx) \vee (gx)]\})]\}$ was false is untenable. It is therefore a valid statement.

There was no branching in the proof above, or at least none worth making explicit, but sometimes there will be, just as there was in zeroeth order logic. As an example, consider the statement

$$\{(\forall x.\{\forall y.[(fx) \supset (fy)]\}) \supset (\{\forall x.(fx)\} \vee \{\forall x.[\neg(fx)]\})\}.$$

Suppose it were false in at least one interpretation. As before this is $(P \supset Q)$ statement and for it to be false we'd need P to be true and Q to be false. So we take $(\forall x.\{\forall y.[(fx) \supset (fy)]\})$ to be true and $(\{\forall x.(fx)\} \vee \{\forall x.[\neg(fx)]\})$ to be false. The second of these statements is of the form $(P \vee Q)$. For it to be false both P and Q must be, so in this case $\{\forall x.(fx)\}$ and $\{\forall x.[\neg(fx)]\}$ must be false. For $\{\forall x.(fx)\}$ to be false there must be an a such that (fa) is false. Similarly, for $\{\forall x.[\neg(fx)]\}$ there must be a b such that $[\neg(fb)]$ is false, and hence such that (fb) is true.

Note that we had to use a new name for this second parameter. It wouldn't have been legitimate to call it a since a was the value that made (fa) false and while we know there are values which make each expression false individually there's nothing to assure us that a single value will make (fa) false and make (fb) true. If you look back at the previous proof you'll see a superficially similar situation, where we first said that if $[\exists x.(fx)]$ is true then

there is some a such that fa and then said that if $(\forall x.\{\neg[(fx) \vee (gx)]\})$ is true then $\{\neg[(fa) \vee (ga)]\}$. I used the same parameter a for both statements. That was legitimate though, since the second statement was about all values and so applies in particular to the value a chosen previously. So in that case I was allowed to reuse the parameter. I wasn't required to though. I could have said that if $(\forall x.\{\neg[(fx) \vee (gx)]\})$ is true then $\{\neg[(fb) \vee (gb)]\}$. That would also have been legitimate, but it wouldn't have led to the contradiction I was looking for.

After that digression let's return to our proof. To summarise where we are, $(\forall x.\{\forall y.[(fx) \supset (fy)]\})$ is true, (fa) is false, and (fb) is true. We use the first of these to conclude that $\{\forall y.[(fb) \supset (fy)]\}$ is true. I'm allowed to reuse the parameter b here because the quantifier is universal and the statement is true. I could also have reused a. That wouldn't have accomplished much though since (fa) is false the statement $[(fa) \supset (fy)]$ wouldn't tell us anything. (fb) on the other hand is true, so the statement $[(fb) \supset (fy)]$ does tell us something. I could also have chosen an entirely new parameter and concluded that $\{\forall y.[(fc) \supset (fy)]\}$. That also wouldn't have accomplished much though, so we'll stick with $\{\forall y.[(fb) \supset (fy)]\}$. From it we can derive $[(fb) \supset (fa)]$. Again I had a choice of parameters and this time I chose a to substitute for y. I could have chosen b instead, or an entirely new parameter. But I didn't. It's at this point that we need to branch the argument since there are two ways that $[(fb) \supset (fa)]$ could be true. (fb) could be false or (fa) could be true. We've already seen though that (fb) is true and (fa) is false. So the assumption that

$$\{(\forall x.\{\forall y.[(fx) \supset (fy)]\}) \supset (\{\forall x.(fx)\} \vee \{\forall x.[\neg(fx)]\})\}$$

is false is seen to be untenable. It is therefore a valid formula.

Tableaux rules for equality and quantifiers

Even though we can't apply the method of truth tables to first order logic we can still apply the method of analytic tableaux. As with zeroeth order logic these are essentially just bookkeeping devices to keep from getting confused in formal arguments like the ones above.

The tableaux rules for the logical operators remain the same but we need new rules for equality and for quantifiers. The rules for equality are relatively straightforward.

First, we can always write a statement of the form $(A = A)$ to the left of the vertical, where A is any parameter. Second, if we have a statement of the form $(A = B)$ to the left of the line, where A and B are parameters, then in any statement we can replace some or all A's by B's or vice versa.

The rules for quantifiers require a bit of notation to describe. In what follows, P is an expression, V is a variable, A is a parameter, B is a parameter which has not appeared previously in the tableau, Q is the expression obtained by substituting the parameter A for all free occurrences of V in P and R is the expression obtained by substituting the parameter B for all free occurrences of V in P. In terms of these we can express the following four rules for the two quantifiers, either of which can appear either to the left or the right of the line.

If we have an expression of the form $(\forall V.P)$ to the left of the line we branch, with one of the branches having a $[\exists V.(V = A)]$ to the right of the line and the other having a Q to the left.

If we have a $(\exists V.P)$ to the left of the line we don't branch and we put a $[\exists V.(V = B)]$ to the left of the line and an R to the left.

If we have a $(\forall V.P)$ to the right of the line we don't branch and we put a $[\exists V.(V = B)]$ to the left of the line and a an R to the right.

If we have a $(\exists V.P)$ to the right of the line then we branch and, with one of the branches having a $[\exists V.(V = A)]$ to the right of the line and the other having a Q to the right.

These may seem obscure but they are relatively easily explained. As with the Boolean operators we just need to examine the ways in which each statement could occur.

For example if $(\forall V.P)$ is true then the statement P holds for any value of the variable V. If A is a parameter then either A is a possible or it isn't. If it isn't then there is no value of V equal to A. If there is then the statement P with each free occurrence of V replaced by an A must be true.

If $(\exists V.P)$ is true then there is a value of V which makes the statement P true. We can give this value a name, but we can't assume that it is equal to any previously named value so we give it the new name B. Since this name corresponds to an actually existing value we have $[\exists V.V = B]$. Since

this value is one that when substituted for V in P gives a true statement we have R.

The other two cases, the ones with statements to the right of the line can be analysed similarly.

The restriction to names which have never appeared in the tableau for the two rules where we made such a restriction is in fact unnecessarily drastic. It would have been enough to require that the parameter does not appear anywhere in that branch rather than in the tableau as a whole.

Example tableaux

The tableaux corresponding to the two informal proofs above are given below.

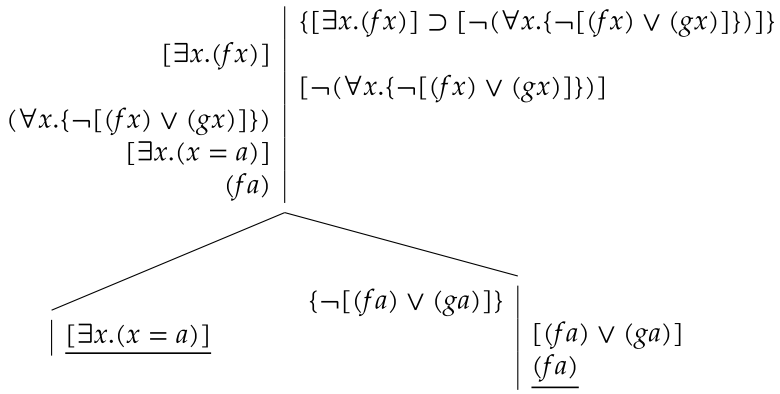


Figure 36: Tableau to check that $[\exists x.(fx)] \supset [\neg(\forall x.\neg[(fx) \vee (gx)])]$ is valid

If you look at the corresponding informal proofs you'll see some extra branching here, reflecting checks on whether various parameters actually refer to anything. They do, since they were introduced as names for things whose existence was asserted by an existential quantifier. In the informal proof this was regarded as so obvious that it didn't require a comment, but in the tableau this gives rise to branches which we are able to close immediately.

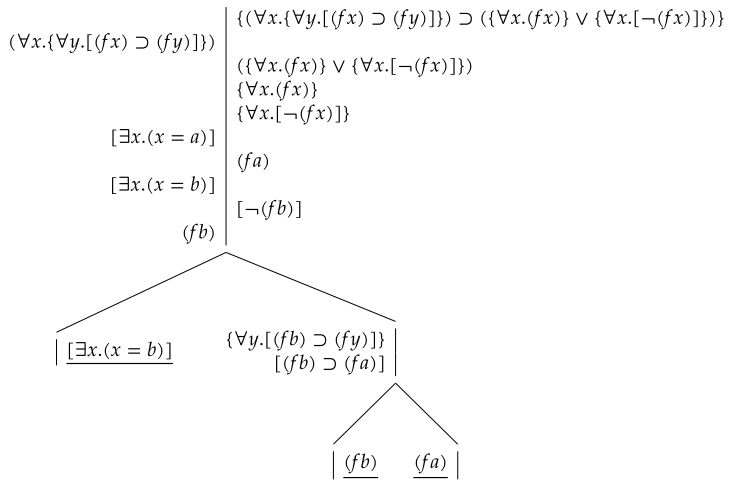


Figure 37: Tableau to check that $(\forall x.\forall y.[(fx) \supset (fy)]) \supset (\forall x.(fx) \vee \forall x.[\neg(fx)])$ is valid

As a third example, consider the tableau proof of

$$[([\forall x.(fx)] \wedge \{\exists x.[(fx) \supset (gx)]\}) \supset (\exists x.\{gx\})].$$

The one corresponds to the following informal argument: For

$$[([\forall x.(fx)] \wedge \{\exists x.[(fx) \supset (gx)]\}) \supset (\exists x.\{gx\})]$$

$[([\forall x.(fx)] \wedge \{\exists x.[(fx) \supset (gx)]\}) \supset (\exists x.\{gx\})]$ to be false in some interpretation $[([\forall x.(fx)] \wedge \{\exists x.[(fx) \supset (gx)]\})]$ would need to be true in that interpretation and $(\exists x.\{gx\})$ would need to be false. Then $[\forall x.(fx)]$ and $\{\exists x.[(fx) \supset (gx)]\}$ would both be true. $\{\exists x.[(fx) \supset (gx)]\}$ asserts the existence of at least one x such that $[(fx) \supset (gx)]$. Let a be such a value of x. Then $[(fa) \supset (ga)]$. We have $[\forall x.(fx)]$, i.e. that (fx) for every value of x. That includes a so (fa). $(\exists x.\{gx\})$ is false, so (ga) is also false. For $[(fa) \supset (ga)]$ to be true we need (fa) to be false or (ga) to be true, but we've already seen that (fa) is true and (ga) is false. The assumption that $[([\forall x.(fx)] \wedge \{\exists x.[(fx) \supset (gx)]\}) \supset (\exists x.\{gx\})]$ is false in some interpretation leads to a contradiction, so it's true in every interpretation, i.e. the statement is valid.

Although the "=" sign appears in the three tableau above we never actually used any of the equality rules for tableau. The example of

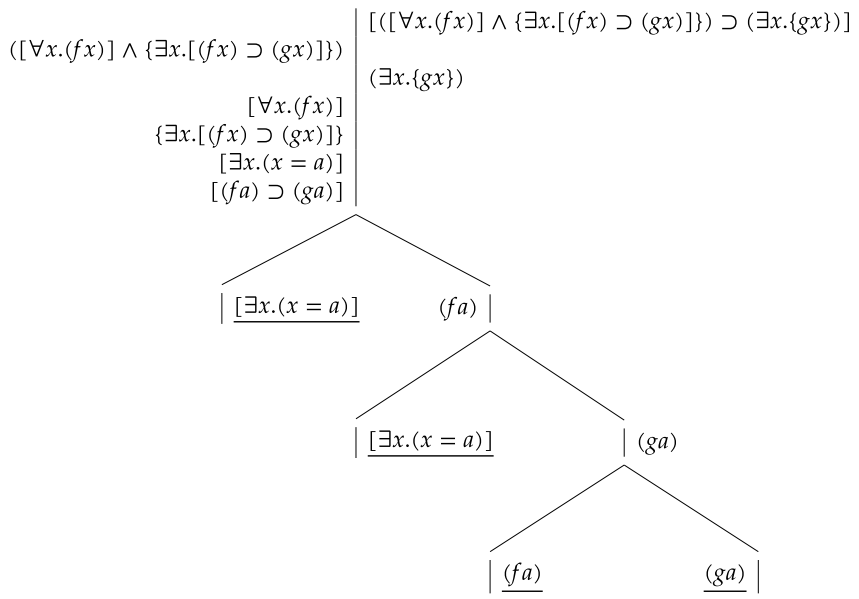


Figure 38: A tableau for $[([\forall x.(fx)] \wedge \exists x.[(fx) \supset (gx)]) \supset (\exists x.gx)]$

$(\forall x.\{\forall y.[(x = y) \supset (y = x)]\})$, i.e. the fact that equality is symmetric, shows them in use. Remarkably, it has no branching at all.

In the second to last line we used the second of our equality rules and the $(a = b)$ on the left to replace the b in in the $(b = a)$ on the right with an a, giving an $(a = a)$ to the right. The first equality rule allows us to write $(a = a)$ on the left as well though, so we obtain our desired contradiction and close the branch.

Tableaux as nondeterministic computations

As in zeroeth order logic the method of analytic tableaux in first order logic can be thought of in terms of nondeterministic computation. There the order in which the rules are applied turned out to be relevant only in the sense that some orders gave an answer faster than others. Here the situation is unfortunately more complicated. There's no guarantee, for example that the method ever terminates. In zeroeth order logic this was guaranteed by two facts: that all tableaux rules result in statements of lower degree than what we start with and that only finitely many–in fact at most

$$
\begin{array}{r|l}
 & (\forall x.\{\forall y.[(x = y) \supset (y = x)]\}) \\
[\exists x.(x = a)] & \\
 & \{\forall y.[(a = y) \supset (y = a)]\} \\
[\exists y.(y = b)] & \\
 & [(a = b) \supset (b = a)] \\
(a = b) & \\
 & (b = a) \\
 & (a = a) \\
\underline{(a = a)} &
\end{array}
$$

Figure 39: A tableau for $(\forall x.\forall y.[(x = y) \supset (y = x)])$

two–statements can be derived from any statement. The first of these is still true for tableaux in first order logic but the second is not. Our rules for quantifiers can be used to derive infinitely many statements from a single one, just by using a different parameter each time.

Even when the tableau can be made to terminate after finitely many steps a poor set of choices can result in it not terminating. In our first example we derived (fa) from $[\exists x.(fx)]$, for example. Instead of proceeding as I did above I could then have derived (fb) and then (fx), etc., never arriving at a contradiction.

The tableaux method for first order logic is more like the parsing problem where we first met nondeterministic computation than it is like the tableaux method for zeroeth order logic. As happened in that problem we can replace the nondeterministic computation by a deterministic one by calculating all paths the nondeterministic calculation could take. As happened there, this deterministic calculation will terminate successfully in finite time if the nondeterministic one could terminate successfully in finite time. Also as happened there, there is no guarantee that it will terminate unsuccessfully in finite time if it can't terminate successfully. It could just run forever.

There is one complication here that we didn't meet in the parsing problem. There there were only finitely many options at each step. Here there will be infinitely many if we are able to apply our quantifier rules, since there are

always infinitely many parameters to choose from. This problem is more apparent than real though. If we choose a new parameter then it doesn't really matter which one we choose. If we choose a different one then the rest of the computation will proceed exactly the same, just with some parameters replaced by others. Whether it terminates, and if so whether it terminates successfully, will remain unchanged. So instead of following all possible substitutions by a parameter we can follow just a single new parameter and, in the two cases where it's allowed, substitution of a parameter already used in the tableau, of which there are only finitely many at each stage.

In this way we obtain an algorithm which is guaranteed to prove any valid statement in finite time. It can prove some invalid statements invalid in finite time as well, but is not guaranteed to do so.

Natural deduction for first order logic

It's possible to extend the natural deduction system we built for zeroeth order logic to first order logic by introducing new rules of inference, to include quantifiers. Writing down sound rules is more difficult than you might expect. At least one textbook, which I will not name, went through multiple editions, each with a different unsound set of rules, before finally finding a correct set.

The rules for equality are relatively straightforward, reflecting the familiar properties of reflexivity, symmetry and transitivity of the = sign.

- For any parameter A we can derive a statement of the form $(A = A)$.
- From any statement of the form $(A = B)$, where A and B are parameters, we can derive $(B = A)$.
- From any statements of the form $(A = B)$ and $(B = C)$, where A, B and C are parameters, we can derive $(A = C)$.
- From any statement of the form $(A = B)$, where A and B are parameters, and P, where P is an expression, we can deduce Q, where Q is the result of replacing one or more occurrences of A in P with B or one or more occurrences of B with A.

In the last rule, note that we aren't required to replace all occurrences, unlike in some of our substitution rules.

It's the rules for quantifiers which tend to cause trouble.

- The expressions $[\neg(\exists V.P)]$ and $[\forall V.(\neg P)]$ are freely interchangeable, where V is a variable and P is an expression.
- The expressions $[\neg(\forall V.P)]$ and $[\exists V.(\neg P)]$ are freely interchangeable, where V is a variable and P is an expression.
- From a statement of the form $(\forall V.P)$ we can deduce $\{[\exists V.(V = A)] \supset Q\}$, where V is a variable, P is an expression, A is a parameter, and Q is the result of replacing all free occurrences of V in P by A.
- From a statement of the form $\{[\exists V.(V = A)] \wedge Q\}$ we can deduce $(\exists V.P)$, where V is a variable, P is an expression, A is a parameter, and Q is the result of replacing all free occurrences of V in P by A.
- The expressions $(\exists V.P)$ and $\{[\exists V.(V = B)] \wedge R\}$ are freely interchangeable, where V is a variable, P is an expression, B is a parameter not appearing in any statement available in the current scope, and R is the result of replacing all free occurrences of V in P by B.
- The expressions $(\forall V.P)$ and $\{[\exists V.(V = B)] \supset R\}$ are freely interchangeable, where V is a variable, P is an expression, B is a parameter not appearing in any statement available in the current scope, and R is the result of replacing all free occurrences of V in P by B.

Besides adding these three rules for equality and six rules for quantifiers we can remove one of our rules of inference from zeroeth order logic, the rule of substitution. It deals with the substitution of expressions for Boolean variables, but we have no Boolean variables in our language for first order logic so we would never be able to apply it. This is the only one of our original rules of inference which referred to Boolean variables so no other rule of inference is similarly affected.

We don't need a replacement for the rule of substitution, but there are two useful rules we can add. Anything which could be proved with them could also be proved without them, but they are sound rules of inference and make some proofs considerably shorter.

- In any expression we can replace all occurrences of any variable with no free occurrences in that expression with any variable not appearing in that expression.
- In any available statement we may substitute any variable for any variable which does not appear in any hypothesis which was active in that statement's scope, provided the substitution doesn't convert free occurrences to bound ones.
- If S is a tautology of zeroeth order logic then we can derive any statement in which replace each Boolean variable in S by an expression in first order logic, provided each occurrence of a Boolean variable is replaced by the same expression.

These mimic some uses of the rule of substitution from zeroeth order logic, to rename variables and to avoid repeating proofs for different substitution instances of the same tautology.

The limitation in the second rule to variables not appearing in active hypotheses is familiar from zeroeth order logic, and is needed for the same reason as there. The requirement that no free instances become bound is new, since we didn't have free and bound occurrences in zeroth order logic. It's there to ensure that we can, for example, deduce

$$(\forall v.\{\forall w.[\exists x.(f v w x)]\})$$

from

$$(\forall x.\{\forall y.[\exists z.(f x y z)]\})$$

but can't deduce

$$(\forall z.\{\forall y.[\exists z.(f z y z)]\})$$

from it. To see why that would be a problem, consider the interpretation of first order logic where the domain is the natural numbers and f is sum relation, if $f x y z$ means $x + y = z$. In this case

$$(\forall x.\{\forall y.[\exists z.(f x y z)]\})$$

means

$$(\forall x.\{\forall y.[\exists z.(x + y = z)]\}),$$

which simply expresses the fact that the sum of any two natural numbers is a natural number. This is a true statement on this interpretation. On the other hand

$$(\forall z.\{\forall y.[\exists z.(fzyz)]\})$$

would mean

$$(\forall z.\{\forall y.[\exists z.(z + y = z)]\}).$$

The inner expression $z + y = z$ means that adding y to z leaves z unchanged, which is true for $y = 0$ but not for any other y. The larger expression $[\exists z.(z + y = z)]$ is therefore also true for $y = 0$. Since it's not true for all y the larger expression $\{\forall y.[\exists z.(z + y = z)]\}$ is false. It's false no matter what natural number is substituted for all free occurrences of z since there are no free occurrences of z in $\{\forall y.[\exists z.(z + y = z)]\}$. The full statement is therefore false. So the rule without this restriction would be unsound, since it would allow us to deduce a false statement from a true one in at least one intended interpretation. The phenomenon we're ruling out is called "variable capture", since it causes a free variable to become bound.

Although we haven't formalised this yet, and can't until we have a proper definition of computability, one of our requirements for a formal system is that questions of whether a statement follows from previous statements by a rule of inference must always be decidable. The second rule above satisfies this requirement because we've already introduced a procedure to check whether a statement in zeroeth order logic is a tautology.

Soundness, consistency and completeness of first order logic

Our natural deduction system for first order logic is sound, in the sense that the axioms are true in any of the interpretations considered earlier and it's not possible to derive a false statement true statements in any of those interpretations using any of our rules of inference. In fact there are no axioms, so that part, at least is easy. Of course if you look closely the statement of soundness above you'll see a few quantifiers appearing explicitly, like "in any of those interpretations" and a few more which are implicit. So any system in which we might hope to prove the soundness of first order logic will have to include first order logic in its foundations, so a formal proof doesn't really accomplish anything. It's still possible to give informal proofs though. If you stare at the rules of inference you should be able

to convince yourself that they are all reasonable. Of course people have convinced themselves of the reasonableness of unsound rules of inference in the past so there's a limit to how far you should trust your intuition in these matters.

Consistency will follow from completeness, so we don't need to consider it separately.

The system can be proved complete by roughly the same method as was used previously for zeroeth order logic. Either we can find a tableau which closes or we can't. If we can then there is an algorithm for converting that tableau into a formal proof, and so the statement is a theorem of the system. Unfortunately though the tableau method doesn't need to terminate, even when presented with a valid statement.

One important property of zeroth order tableaux, that every statement is shorter than the statement it was derived from, no longer holds. We might get stuck performing tableau operations forever without ever completing the tableau. We can still think of the infinite tableau that would result from this procedure though, even though we can't produce it in finite time. A further complication comes from the variety of choices we have to make at each step. If we choose particularly badly we could continue forever even though other choices might cause the tableau to close. This is a solvable problem though. The idea, roughly, is to view the tableau method as a non-deterministic computation and use one of the hybrid traversal methods to ensure that if some set of choices has a desired property, like causing the tableau to close, then we will eventually find it. With a bit more care we can also ensure that in the case where it doesn't close the open branches, which may be infinite, contain enough information to generate a counter-example, i.e. an interpretation which makes the statement we started with false. In that case the statement must be invalid.

The above arguments show that every statement is a theorem or is invalid. Equivalently, every valid statement is a theorem, which is completeness.

Elementary arithmetic

Logic has been described as "the subject in which nobody knows what one is talking about, nor whether what one is saying is true." This means in

logic we don't analyse the content of statements, or even have a way of expressing that content, we're just concerned with how those statements are connected.

It's time to start making statements with actual content. We'll do this in two settings, elementary arithmetic and set theory. We'll start with elementary arithmetic because it's more familiar, although formal proofs in elementary arithmetic may not be.

A language for arithmetic

Our language for this is given by the grammar

```
statement : bool_exp
bool_exp : bndd_exp
      | "(" "¬" bool_exp ")" | "(" bool_exp b_operator bool_exp ")"
         | "(" quantifier variable "." bool_exp ")"
         | "(" quantifier variable "<" bound ":" bool_exp ")"
      | "[" "¬" bool_exp "]" | "[" bool_exp b_operator bool_exp "]"
         | "[" quantifier variable "." bool_exp "]"
         | "(" quantifier variable "<" bound ":" bool_exp "]"
      | "{" "¬" bool_exp "}" | "{" bool_exp b_operator bool_exp "}"
         | "{" quantifier variable "." bool_exp "}"
         | "{" quantifier variable "<" bound ":" bool_exp "}"
         | comparison
bndd_exp : "(" "¬" bndd_exp ")" | "(" bndd_exp b_operator bndd_exp ")"
         | "(" quantifier variable "<" bound ":" bndd_exp ")"
      | "[" "¬" bndd_exp "]" | "[" bndd_exp b_operator bndd_exp "]"
         | "(" quantifier variable "<" bound ":" bndd_exp "]"
      | "{" "¬" bndd_exp "}" | "{" bndd_exp b_operator bndd_exp "}"
         | "{" quantifier variable "<" bound ":" bndd_exp "}"
         | comparison
comparison : "(" num_exp c_relation num_exp ")"
           | "[" num_exp c_relation num_exp "]"
           | "{" num_exp c_relation num_exp "}"
b_operator : "∧" | "∨"  | "⊃"
quantifier : "∀" | "∃"
variable : letter | variable "!"
```

```
variable : "v" | "w" | "x" | "y" | "z"
c_relation : "=" | "<" | ">" | "≤" | "≥"
bound : const_exp | variable
num_exp : bound | num_exp "'"
        | "(" num_exp a_operator num_exp ")"
        | "[" num_exp a_operator num_exp "]"
        | "{" num_exp a_operator num_exp "}"
const_exp : "0" | const_exp "'"
          | "(" const_exp a_operator const_exp ")"
          | "[" const_exp a_operator const_exp "]"
          | "{" const_exp a_operator const_exp "}"
a_operator : "+" | "-" | "·"
```

Interpretation

The logical symbols, i.e. the Boolean operators and quantifiers, have the same meaning as in logic. There are two types of quantifiers though. The bounded quantifiers are just a useful shorthand.

$$\{\forall x < c : P\}$$

means the same as

$$\{\forall x.[(x < c) \supset P]\}$$

and similarly for the existential quantifier.

The arithmetic operators $+$ and $-$ mean addition and multiplication, respectively. 0 means 0. The apostrophe is the successor operator, that is the operator which takes a natural number and increments it. The successor operator occurs so often that I've broken with my usual practice of allowing only fully parenthesised expressions. This won't cause any problems because it's the only exception.

This language doesn't allow the usual decimal notation for natural numbers so the way to represent the natural numbers we'd normally call 0, 1, 2, 3, ... is as 0, 0′, 0′′, 0′′′, etc. We can also add and multiply numerical expressions, which gives us a few more options. So we don't have to represent 2023 as a 0 followed by 2023 apostrophes, for example. We could also write it as

$$(0''' + \{[0' + (\{0'' + [(0''' + \{[0''' + (0' \cdot 0'''')] \cdot 0''''\}) \cdot 0''''] \} \cdot 0'''') \cdot 0'''')] \cdot 0''''\}).$$

If you're wondering where this came from it's just the base 4 representation

$$(3 + \{[1 + (\{2 + [(3 + \{[3 + (1 \cdot 4)] \cdot 4\}) \cdot 4]\} \cdot 4)] \cdot 4\})$$

with 1, 2, 3 and 4 replaced by $0'$, $0''$, $0'''$, and $0''''$. There's nothing special about 4. We could have used decimal instead but it's hard to look at $0'''''''''$ and see what number it is. We could also have used binary but then the expression would be quite long, though not nearly as long as a 0 followed by 2023 apostrophes!

Note that some expressions are of numerical type, which in this case means they should be though of as natural numbers, and some are of Boolean type.

Numerical expressions include `const_exps`, `variables` and `num_exps`. `const_exps`, i.e. constant expressions, are those constructed without the use of `variables`. Every `const_exp` is a `num_exp`, but not every `num_exp` is a `const_exp`. The expression

$$(0''' + \{[0' + (\{0'' + [(0''' + \{[0''' + (0' \cdot 0'''')] \cdot 0''''\}) \cdot 0''''] \} \cdot 0'''') \cdot 0'''')] \cdot 0''''\}).$$

for 2023 considered earlier is a `const_exp`, and therefore also a `num_exp`. $(x + y')$ is a `num_exp` which is not a `const_exp`.

Boolean expressions include `comparison`, `bndd_exp` and `bool_exp`. Every `comparison` is a `bndd_exp` and every `bndd_exp` is a `bool_exp` but not every `bool_exp` is a `bndd_exp` and not every `bndd_exp` is a `comparison`. `comparisons` have to be constructed without the use of Boolean operators or quantifiers. `bndd_exps` can use Boolean operators and quantifiers, but only bounded quantifiers.

The point of bounded expressions is that for any given value of the free variables occurring in them we can, at least in principle, check whether they're true. Whenever we see a quantifier we only need to check those numbers up to the given bound. With an ordinary quantifier we might need to check all natural numbers, which cannot be done in finite time.

It would have been possible to use decimal or binary representations as part of the language but then we'd have build much of elementary arithmetic into the axioms and rules of inference. This can be done, as we saw when we considered languages expressing divisibility properties. In addition to requiring a very complicated set of rules of inference that approach

would miss the point. We want to build a formal system in which to prove statements in elementary arithmetic. If we need to assume large parts of elementary arithmetic to show that our rules of inference are sound then what's the point? It's better to assume as little prior knowledge of arithmetic as we can get away with.

Note that there Boolean operators, which combine Boolean expressions to give a Boolean expression, and there are arithmetic operators, which combine numerical expressions to give a numerical expression, and there are comparison relations, which combine numerical expressions to give a Boolean expression, but there is no way of combining Boolean expressions to get a numerical expression.

Redundancy and ambiguity

There is some redundancy in this language. As already mentioned, the bounded quantifier expressions are just shorthand for longer expressions involving ordinary quantifiers.

Also we would suffer no loss of expressiveness if we removed the $\leq$ relation, for example. The statement

$$(x \leq z),$$

for example, has the same meaning as

$$\{\exists y.[(x + y) = z]\},$$

since $x \leq z$ if and only if there is a natural number y such that $x + y = z$. Alternatively, we could remove $=$ and write

$$(x = z)$$

as

$$[(x \leq z) \wedge (z \leq x)].$$

In fact it's possible to express any four of our relations in terms of the remaining one.

We also have more non-terminal symbols in our grammar than are strictly necessary to specify this language. We don't really need to consider constant expressions separately from other numerical expressions, for example, or bounded expressions separately from other Boolean expressions.

In fact you can safely ignore those distinctions for most of this chapter, but it will sometimes be useful to have them available.

One thing conspicuously missing from our language is any notation for sets of natural numbers. We can still talk about at least some sets, by specifying conditions for membership, but we can't name sets, we can't assert the existence of a set with given properties and we can't assert that all sets have a given property. This is the "elementary" in "elementary arithmetic".

The grammar above is ambiguous, but not very ambiguous. For example $(0'' + 0'')$, i.e. $2 + 2$, is a `num_exp`. We can see this either by using the fact that it's a `const_exp` and every `const_exp` is a `num_exp`, or by using the fact that `0''` is a `num_exp` and that joining two `num_exps` with a `+` gives a `num_exp`. Strictly speaking these are different parsings of the expression but they have the same meaning and there won't be any cases where it matters which parsing we choose. It is possible to disambiguate the grammar, but it doesn't really seem worthwhile.

A more minimalist version of the language, while it would make proving theorems in our formal system more painful, would be useful if were going to prove a number of theorems about it, but I'm rarely going to do that, although I will state a number of them and give a rough idea of the proof of some.

Expressing more complex ideas

All of elementary arithmetic can be expressed in this language, but sometimes a bit of ingenuity is required. We can, for example, compensate for the lack of subtraction and division signs. $x = z - y$ can be expressed as $[(x + y) = z]$. The second statement implies $(y \leq z)$, without which the first wouldn't make sense. Similarly, $x = z/y$ can be expressed as $[(x \cdot y) = z]$.

Knowing that statements about division can be expressed via statements about multiplication we can see how to express divisibility. The condition that z is divisible by x, i.e. that x is a divisor of z, for example, can be expressed as $\{\exists y.[(x \cdot y) = z]\}$.

We can also express primality. The following sentence is one way of saying

that z is prime:

$$\{[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})] \wedge (0' < z)\}.$$

As with all statements, this one is best understood by breaking it into smaller phrases. Starting with

$$[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})]$$

we can peel off the universal quantifiers and and ask when

$$\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\}$$

is true. This means if $[(x \cdot y) = z]$, i.e. if z is the product of x and y, then $[(x = z) \vee (y = z)]$, i.e. at least one of x or y is equal to z. Since the only way to write a prime as a product of natural numbers is 1 times itself, in either order, the statement

$$[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})]$$

is true whenever z is prime. There are two non-prime values of z for which the statement above is true though, 0 and 1. To exclude these we add the additional condition

$$(0'' \leq z),$$

which ensures that z is greater than 1.

We can express even more complicated thoughts. We can say, for example, that there are infinitely many primes. It's not immediately obvious how to do this. We've just seen how to express the fact that any particular number is prime but how can we make a statement about infinitely many numbers in language which doesn't have a notation for sets or infinity? There is a standard trick for this. To say that there are infinitely many primes we say that for every number w there is a prime number z greater than w. In our language this is

$$\{\forall w.[\exists z.(\{w < z\} \vee \{[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})] \wedge (0' < z)\})]\}.$$

As you can see, our expressions are starting to get unwieldy. It's helpful to introduce a notational convention. I'll use capital letters to refer to substitution instances of particular expressions and follow those letters with the particular expressions to be substituted, e.g. I'll write

$$D(x, z) \equiv \{\exists y.[(x \cdot y) = z]\}$$

to mean that D followed by an open parenthesis, then a numerical expression, then a comma, then another numerical expression, represents the expression obtain by substituting the first numerical expression for the x and the second numerical expression for z in the Boolean expression $\{\exists y.[(x \cdot y) = z]\}$. Although this is the literal meaning of the notation, once we understand the expression defining D we think of $D(x, z)$ simply as "x divides z". Note that neither D nor $\equiv$ is part of our language for elementary arithmetic. These are part of a notation for talking about the language of elementary arithmetic. Similarly, we could introduce the shorthand

$$P(z) \equiv \{[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})] \wedge (0' < z)\}$$

to express primality and then write the statement that there are infinitely many primes as

$$\{\forall w.[\exists z.(\{w < z\} \vee P(z))]\}.$$

This isn't part of the language of elementary arithmetic, since P isn't part of that language, but it uniquely identifies a statement which is part of the language, namely the one given previously to express the same meaning.

Arithmetic subsets and relations

Some subsets of the natural numbers can be described by statements in one or the other of our two languages. As we just saw, the set of prime numbers can be expressed by such a statement. Some cannot be described by any statement in our language though. A simple proof of this fact will be presented in the set theory chapter. A set which can be described by a statement is called arithmetic. The accent is on the third syllable, in contrast to its use in phrases like "elementary arithmetic", where the accent is on the second syllable.

Note that sets of numbers are not part of our language. The closest we have is Boolean expressions with one free variable. We can think of the set of values of the variable which make that expression true but we can't assign a name to that set within our language.

An example of an arithmetic set is the set of powers of 2. Our language doesn't have any notation for exponentiation so we can't just say z is a power of 2 if there is some y such that $z = 2^y$. Instead we can observe that if z is a power of 2 then every divisor of z is either 1 or is a multiple of 2, and

conversely that every z with this property is a power of 2. This is something we can translate into our language. "x is a divisor of z" translates as

$$\{\exists y.[(x \cdot y) = z]\}.$$

"x is 1" is just

$$(x = 0').$$

"x is a multiple of 2" is

$$\{\exists y.[x = (0'' \cdot y)]\}.$$

So "every divisor of z is either 1 or a multiple of 2" translates as

$$[\forall x.(\{\exists y.[(x \cdot y) = z]\} \supset [(x = 0') \vee \{\exists y.[x = (0'' \cdot y)]\}])].$$

Another example of an arithmetic set is the set of Fibonacci numbers, although proving this will require more work. Let f_n be the n'th Fibonacci number, defined recursively by

$$f_0 = 0, \quad f_1 = 1, \quad f_{n+2} = f_n + f_{n+1}.$$

For $n = 1$ we have

$$f_{n+1}^2 = f_n f_{n+2} + (-1)^n.$$

Suppose that the equation above holds for $n = m$, i.e. that

$$f_{m+1}^2 = f_m f_{m+2} + (-1)^m.$$

Then

$$\begin{aligned} f_{m+3} f_{m+1} &= (f_{m+2} + f_{m+1}) f_{m+1} \\ &= f_{m+2} f_{m+1} + f_{m+1}^2 \\ &= f_{m+2} f_{m+1} + f_m f_{m+2} + (-1)^m \\ &= f_{m+2}(f_{m+1} + f_m) + (-1)^m \\ &= f_{m+2} f_{m+2} + (-1)^m \\ &= f_{m+2}^2 - (-1)^{m+1} \end{aligned}$$

and therefore

$$f_{m+2}^2 = f_{m+3} f_{m+1} + (-1)^{m+1}.$$

So

$$f_{n+1}^2 = f_n f_{n+2} + (-1)^n$$

is true also when $n = m + 1$. Since it holds for $n = 0$ and holds for $n = m + 1$ whenever it holds for $n = m$ it must hold for all n, by induction. Now

$$f_{n+2} = f_n + f_{n+1}$$

so we can rewrite our relation above as

$$f_{n+1}^2 = f_n(f_n + f_{n+1}) + (-1)^n.$$

Consider the case where n is even, i.e. $n = 2k$, and let

$$x_k = f_{2k}, \quad y_k = f_{2k+1}.$$

Then the equation above becomes

$$y_k = x_k(x_k + y_k) + 1.$$

Suppose z is a Fibonacci number. Then $z = x_k$ or $z = y_k$ for some value of k. So, in our language for elementary arithmetic,

$$[\exists x.(\exists y.\{(y \cdot y = \{[x \cdot (x + y)] + 1\}) \wedge [(z = x) \vee (z = y)]\})].$$

We've just seen that if z is a Fibonacci number then it makes this statement, with our usual interpretation, true. The converse is true as well, although that's even harder to prove, and I'll skip this part. So the Fibonacci numbers are described by a statement in our language and so are an arithmetic set.

In addition to arithmetic sets we can also talk about arithmetic relations. These correspond to Boolean expressions with multiple free variables. In fact we've seen a few examples already. "... is divisible by ..." is an arithmetic relation.

Bounded arithmetic

There is something a bit unsatisfying about the arguments which showed that the powers of two and the Fibonacci numbers are arithmetic. For one thing, we seem to be putting more arithmetic into our language than we are getting out of it. Both sequences are very easy to define but in order to show that their elements are an arithmetic set we needed to borrow some facts from number theory which are considerably deeper than we'd need for the

definitions. The other problem is that it's not clear how to generalise those arguments to other simple sequences, even very closely related ones. We could, for example, use the argument above with only very minor changes to show that the powers of 3 or of 5 are arithmetic sets. In fact, for any prime p the expression

$$[\forall x.(\{\exists y.[(x \cdot y) = z]\} \supset [(x = 0') \vee \{\exists y.[x = (p \cdot y)]\}])]$$

says that z is a power of p.

For powers of 4 we can use the fact that a number is a power of 4 if and only if it is the square of a power of 2. What about powers of 6, though? While it's true that every power of 6 is the product of a power of 2 and a power of 3 it's not true that every such product is a power of 6.

It would be nice to have a general principle saying that given an initial value and an arithmetic relation which uniquely determines the next element of the sequence from the current one the set of all elements of the sequence is arithmetic. This would give us a more satisfactory proof that the powers of 2 are an arithmetic set, and indeed that any geometric progression is arithmetic. With a bit more work it could also give us a more satisfactory proof that the set of Fibonacci numbers is arithmetic. This turns out to be too much to ask for, but there is a weaker principle which is true and suffices for most applications.

Consider the set of bounded expressions. Every such expression is a Boolean expression. We defined sets and relations to be arithmetic if they were defined by Boolean expressions. We can similarly describe sets or relations to be constructively arithmetic if they are defined by bounded expressions. Then every constructively arithmetic set or relation is arithmetic. Is the converse true? It might seem obvious that it's not, since there are Boolean expressions which are not bounded. That's not a valid argument though, since the same set or relation might be defined by more than one expression, and some might be bounded while others might not be. For example, our primality criterion

$$[\forall x.(\forall y.\{[(x \cdot y) = z] \supset [(x = z) \vee (y = z)]\})]$$

had two ordinary, unbounded, quantifiers, but there are other expressions which identify whether z is prime which use only bounded quantifiers. z is

prime if and only if it is greater than 1 and cannot be written as the product of two numbers less than z. In other words,

$$\{(z > 0') \wedge [\forall x < z : (\forall y < z : \{\neg[(x \cdot y) = z]\})]\}.$$

Now all the quantifiers are bounded, so the set of primes is not just arithmetic but also constructively arithmetic. It's useful to know that primality is a constructively arithmetic property, so I'll now replace our earlier primality test with this one:

$$P(z) \equiv \{(z > 0') \wedge [\forall x < z : (\forall y < z : \{\neg[(x \cdot y) = z]\})]\}.$$

Even though the argument suggested above for the existence of sets which are arithmetic but not constructively arithmetic turned out to be invalid its conclusion is nonetheless true. There are sets which are arithmetic but not constructively arithmetic.

Now that we have the notion of constructively arithmetic sets and relations we can state our substitute principle. Given an initial value and a constructive arithmetic relation which uniquely determines the next element of the sequence from the current one the set of all elements of the sequence is arithmetic. I won't prove this though.

Encoding

Suppose we want to encode lists of symbols as natural numbers. We don't necessarily need the set of symbols to be finite but it makes things easier and that special case is sufficient for our present purposes. For applications to languages I'm going to assume that there is only one token per terminal symbol. If the set of tokens is finite then we can always arrange this, but also it happens to be true already for most of the languages we'll consider, including the language of elementary arithmetic considered in this chapter.

One tempting idea, if we have b tokens, is to use a base b encoding, where we assign one digit to each symbol and just convert a list of symbols to the corresponding list of digits, and then convert that to the natural number whose base b representation is that list of digits. There's a subtle problem with this idea though, which is that some symbol will be assigned the digit 0 and two lists which differ only in the number of occurrences of that symbol at the start of the list will give lists of digits differ only in the number

of leading 0's and will therefore be encoded as the same natural number. This is therefore a lossy encoding.

There's an easy fix to the problem above. We can use a base $b+1$ encoding, and just not use the digit 0 for any of the symbols. This isn't a terrible idea but it does have one disadvantage. Unlike the previous encoding, not every natural number will correspond to to a string of symbols. Only those natural numbers whose base $b+1$ encoding have no 0 digits will encode lists of symbols. That's better than having some natural number represent multiple lists, but it's still somewhat inconvenient.

Following an idea of Raymond Smullyan we can modify the base b encoding as follows. Suppose that, just as with ordinary base b, we associate the list of digits

$$(d_0, d_1, \dots, d_{l-2}, d_{l-1})$$

with the natural number

$$d_0 \cdot b^{l-1} + d_1 \cdot b^{l-2} + \cdots + d_{l-2} \cdot b + d_{l-1}$$

but instead of choosing the digits from the set $\{0, 1, \dots, b-2, b-1$ we choose them from $\{1, 2, \dots, b-1, b\}$. Like the base $b+1$ encoding above this encoding is lossless, since it's fairly easy to show that no two lists give the same natural number. It still has the minor problem that not every natural number is the encoding of a list, but this time the only number which is left out is 0, a problem which we can easily fix by subtracting 1 from the expression above. So our new encoding is to assign the digits $\{1, 2, \dots, b-1, b\}$ to our symbols, convert the list of symbols to a list of digits, and then form the natural number

$$d_0 \cdot b^{l-1} + d_1 \cdot b^{l-2} + \cdots + d_{l-2} \cdot b + d_{l-1} - 1.$$

Now every list of symbols is represented by a natural number and every natural number represents a list of symbols.

Once we know how to encode lists of symbols as natural numbers we can encode sets of lists of symbols as sets of natural numbers, relations on lists of symbols as relations on natural numbers, etc.

As an example, consider the language of balanced parentheses. Our original grammar was

```
ok : | "(" ok ")" ok
```

For reasons which will become clear soon I want to allow not just strings with balanced parentheses by finite sequences of such strings, separated by commas, so our new language is

```
seq : ok | seq "," ok
ok  : | "(" ok ")" ok
```

Strings with balanced parentheses can then be thought of as sequences with only one element. This new grammar has five symbols, the three terminals `(`, `)` and `,` and the two non-terminals `seq` and `ok`. We can assign those five symbols to the five base five digits 1, 2, 3, 4 and 5, in that order. Then the string `(()(()))`, for example, would be encoded as

$$1 \cdot 5^7 + 1 \cdot 5^6 + 2 \cdot 5^5 + 1 \cdot 5^4 + 1 \cdot 5^3 + 2 \cdot 5^2 + 2 \cdot 5 + 2 - 1$$

This is the decimal number 100811, but there really isn't much reason to convert these encodings to and from decimal.

The list of symbols in the example above happened to be a member of the language, in fact a member of both the original language and the extended one, but this encoding scheme allows us to represent any list of symbols as a natural number. This raises an obvious question: can we identify which natural numbers correspond to members of the language? There are actually two languages here, the original language of strings with balanced parentheses and the extended language of sequences of such strings. Although the answer turns out to be the same for both the question we're really interested in is the one for the original language.

The encoding system described above has the very useful property that the concatenation relation is arithmetic, and in fact constructively arithmetic. If you're not familiar with concatenation, the concatenation of the list

$$(c_0, c_1, \dots, c_{k-2}, c_{k-1})$$

and the list

$$(d_0, d_1, \dots, d_{l-2}, d_{l-1})$$

is the list

$$(c_0, \dots, c_{k-1}, d_0, \dots, d_{l-1}).$$

Suppose x is the encoding of the first list above, y is the encoding of the second list, and z is the encoding of their concatenation, i.e. third list. In other words, suppose that

$$x' = c_0 \cdot b^{k-1} + c_1 \cdot b^{k-2} + \cdots + c_{k-2} \cdot b + c_{k-1},$$

$$y' = d_0 \cdot b^{l-1} + d_1 \cdot b^{l-2} + \cdots + d_{l-2} \cdot b + d_{l-1},$$

and

$$z' = c_0 \cdot b^{k+l-1} + \cdots + c_{k-1} \cdot b^l + d_0 \cdot b^{l-1} + \cdots d_{l-1}.$$

Then

$$z' = b^l \cdot x' + y'.$$

Aside from the parentheses, which we can easily add, the thing which prevents this from being an expression in our language is the power b^l. At this point it is helpful to make the additional assumption that b is prime. This is avoidable but avoiding it requires considerable effort and in all the applications we have in mind we have the option of adding more symbols and simply not using them so there is no real loss of generality. Of course what we're using here is the fact that for every natural number there is a larger number which is prime, i.e. that there are infinitely many primes, mentioned earlier. In this case z is a power of b if and only if every divisor of z is either 1 or b. Suppose z is positive. Then we only need to test divisors between 1 and z, since there are no others. x is a divisor if and only if there is a y such that $x \cdot y = z$. Again, if z is positive then we only need to test values of y between 1 and z. So $x = v'$ and $y = w'$ for some natural numbers v and w less than z. The condition that $x = 1$ is equivalent to $v = 0$. The condition that x is a multiple of b is that there is a y such that $b \cdot y = x$. Again, we only need to consider values of y between 1 and x, which are therefore between 1 and z. Again, since y is positive it must be of the form $y = w'$ for some w. In this way we are led to the bounded expression

$$\begin{aligned} Q(z) \equiv \{(z > 0) \wedge [\forall v < z : (\{\exists w < z : [(v' \cdot w') = z]\} \\ \supset [(v = 0) \vee \{\exists w < z : [(b \cdot w') = v']\})]\}. \end{aligned}$$

If you compare this to the earlier expression for prime powers you'll see that in addition to replacing p with b I've added the explicit condition $z > 0$ and I've used v and w instead of x and y. The second change isn't a simple substitution, since the relations between these are $x = v'$ and $y = w'$. Most

importantly, I've bounded all the quantifiers. The trick of using v and w instead of x and y was used to get variables which are strictly less than z, rather than merely less than or equal to it, as required for bounded quantifiers. This also required ruling out the case $z = 0$ explicitly, because there is no upper bound on the divisors of 0.

y' is a number whose modified base b representation has length l. The smallest such number is the one where all the digits are 1, i.e.

$$1 \cdot b^{l-1} + 1 \cdot b^{l-2} + \cdots + 1 \cdot b + 1 = \frac{b^l - 1}{b - 1}$$

The largest such number is the one where all the digits are b, i.e.

$$b \cdot b^{l-1} + b \cdot b^{l-2} + \cdots + b \cdot b + b = b\frac{b^l - 1}{b - 1}$$

In other words,

$$\frac{b^l - 1}{b - 1} \leq y'$$

and

$$y' \leq b\frac{b^l - 1}{b - 1}$$

This is equivalent to

$$b^l + y \leq by'$$

and

$$by + b + b \leq b \cdot b^l + y'.$$

In other words, $w = b^l$ if and only if w is a power of b and satisfies the inequalities

$$w + y \leq by'$$

and

$$by' + b \leq b \cdot w + y'.$$

So b^l is the unique w which makes the expression

$$S(w, x, y, z) \equiv [Q(w) \wedge ([(w + y) \leq (b \cdot y')] \wedge \{[(b \cdot y) + b] \leq [(b \cdot w) + y']\})]$$

true. Now we can express the condition for z to be encoding of the concatenation of the lists encoded by x and y.

$$C(x, y, z) \equiv \{\exists w < y : [S(w, x, y, z) \wedge \{[(w \cdot x') + y'] = z'\}]\}.$$

Encoding grammar

Grammars are defined mostly in terms of concatenation. Consider, for example, our grammar for balanced parentheses. A string belongs to the grammar of balanced parentheses if and only if we can parse it using the rule

```
ok  : | "(" ok ")" ok
```

The string, `(()(()))`, for example, can be parsed in the following steps.

```
ok
( ok ) ok
( ( ok ) ok ) ok
( ( ) ok ) ok
( ( ) ( ok ) ok ) ok
( ( ) ( ( ok ) ok ) ok ) ok
( ( ) ( ( ) ok ) ok ) ok
( ( ) ( ( ) ) ok ) ok
( ( ) ( ( ) ) ) ok
( ( ) ( ( ) ) )
```

This is a sequence of lists of symbols in the grammar, i.e. an element of our extended grammar, except that I've used newlines instead of commas and I've inserted some spaces to make things look nicer. This sequence has four crucial properties.

- The first list has only one element, the start symbol.
- The last list is the one we were trying to parse.
- The last list consists only only of terminal symbols.
- Each list is obtained by taking a previous list and replacing an `ok` by one of its two allowed expansions.

Each of these properties can be expressed in terms of concatenation. Our first task is to separate the one large list of symbols that we have into smaller lists, separated by our separator, which I'm going to refer to as the comma, even though I used newlines above. Note that being a sublist of another list is a property we can describe in terms of concatenation. β is a sublist of δ if we can write δ as the concatenation of α, β and γ. A list contains a comma if and only if and only if it has a sublist with a single element, a comma. The pieces we want to split our list into are the largest comma-less

pieces, i.e. those which have no comma but are not sublists of any other comma-less list. Having split the list in this way it's easy to express the first and last condition above. The second one can be handled similarly to the problem of identifying comma-less lists before. We just need to do that, restricted to the final part of the list, with each non-terminal in place of a comma. The third one isn't much harder. It's straightforward to express in terms of concatenation the fact that one of these maximal lists appears before another in the sequence. β precedes δ if the whole list can be written as the concatenation of α, β, γ, δ and ε. We can also express the fact that δ is obtained from β by expanding an `ok` to an empty list. That means that there are κ, λ, μ, and ν such that β is $\kappa\lambda\nu$, δ is $\kappa\mu\nu$, λ consists of a single `ok`, and μ is empty. Expressing the fact that δ is obtained from β by expanding an `ok` to a list of the form `( ok ) ok` is similar, but we use that list of four symbols for μ. We've now expressed all the conditions above in terms of quantifiers, Boolean operators, and concatenation.

We can now use our earlier observation that concatenation is represented by a constructively arithmetic operation to express parsing in arithmetic terms. Specifically we can express the fact that a given sequence is a valid parsing for a given list of terminal symbols as an arithmetic relation between their encodings. In fact this relation is constructively arithmetic. To see this it suffices to observe that all the lists we need to consider are sublists of the given parsing list and that sublists have encodings which are no larger than the encoding of the full list. The property of belonging to the list of balanced parentheses is also an arithmetic property, since that just means there exists some natural number which encodes a valid parsing, so we just need to stick another quantifier on to whatever expression we've constructed to express the relation of being a valid parsing for a list. There's one complication though. We don't have any bound for the encoding of the parsing in terms of the encoding of the original list, so this won't be a bounded quantifier and we can only conclude that the encodings of members of the language form an arithmetic set. We don't gain any information about from this about whether it's constructively arithmetic, but that's less important.

Encoding non-deterministic computations

How far can the ideas of the previous section be pushed?

Hopefully it's clear that we could do the same thing with any grammar of the type we've been considering so far. With an encoding of this type the set of encodings of members of the language will always be an arithmetic set. In fact we could allow more complicated grammars, but won't pursue this idea for now.

Parsing can, as discussed earlier, be considered as an example of a non-deterministic computation. From this point of view the procedure we've just employed is

- Develop a language which allows us to describe the possible states of the computation.
- Extend this language to encompass possible computational paths.
- Apply an encoding of the type described above to the extended language.
- Create an expression to check whether the initial state is one of the allowed ones.
- Create an expression to determine whether the final state is of the type needed for successful termination of the computation.
- Create an expression to check whether the final state is one is the desired one.
- Create an expression to check whether the each state is reachable from the previous one via one of the allowed actions in that state.
- Combine the above expressions to get an expression which determines, for any state, whether there is a possible computational path which terminates successfully in this state.
- Conclude that the set of all possible final states is an arithmetic set.

This is all fairly straightforward, except for the last step, which will only work for certain types of actions. As we've seen though, the fact that concatenation is constructively arithmetic allows us to perform fairly sophisticated pattern matching operations, so in practice the class of sets we can prove are arithmetic by this technique is quite large.

If you compare this general procedure to the special case of the balanced parentheses example you may notice one way in which it doesn't quite fit. There I described checking that each element of the sequence is obtainable by expanding the token `ok` in some previous list. The recipe given above would require it to expand a token from the immediately previous list. Changing it to conform to the recipe above is possible. If there's a

parsing of the more lenient type considered earlier than there's one of this more restricted type as well. The possibility to use past states as well as the current one in determining the next state is useful though. In this case it would allow us to write a backtracking parser, i.e. one which can explore branches of the state tree tentatively without firmly committing to them. We don't even really need to limit ourselves to examining just one past state in determining what we're allowed to do next. The freedom to make use of more than one previous state would be useful, for example, in showing that the Fibonacci numbers are an arithmetic set, since each number in the sequence depends on two previous ones. This freedom will soon be useful in other contexts. There's no problem with allowing it. Our method uses concatenation to identify previous states, and that's just as easy to do, and in fact slightly easier, if we allow our rules to consider all earlier states and not just the preceding one.

Encoding formal systems

We've also seen that proofs in formal systems can be considered as a type of non-deterministic computation. We can therefore try to use the recipe on them. This will work if the axioms and rules of inference are of an appropriate type. We have to be able to write expressions which check whether a number is the encoding of an axiom of the system, and whether a number is the encoding of a statement which follows, via the rules of inference of the system, from one or more other statements whose encoding is given. The pattern matching capabilities of concatenation make this fairly straightforward for most systems though. A rule like "From statements P and Q we can deduce the statement $(P \wedge Q)$. Also, from any statement of the form $(P \wedge Q)$ we can deduce the statement P and the statement Q'', for example, is very easily expressed. Although it requires considerably more effort we can also express fairly complicated rules, like the rules for quantifiers in our system for first order logic, including their restrictions in certain cases to substituting only new parameters for variables.

In this way we can show that the encoded statements of theorems of first order logic form an arithmetic set. I didn't exactly make this easy though. We don't have any axioms but we have a lot of rules of inference and some of them are fairly complex. The task would be much easier with a more traditional axiomatic system, like a first order version of Nicod. This is

why those systems aren't merely a historical relic. Although it's painful to prove things in them, it's easier to prove things about them.

Encoding arithmetic in arithmetic

We know how to encode expressions in any formal language, or at least any language with only finitely many symbols, in the formal language of elementary arithmetic. The language of elementary arithmetic is a formal language with only finitely many symbols. It follows that we can encode arithmetic in arithmetic! Should we?

Encoding arithmetic in arithmetic can get somewhat confusing. Consider a natural number. It can be expressed in the language of arithmetic. In fact there are multiple ways to express it, but to limit confusion as much as possible let's restrict ourselves to its representation as a 0 with apostrophes. Then there is a unique expression in the language for each natural number. The expression has an encoding, which is a natural number. It's not the same natural number we started with though. It's generally much larger.

At this point I should perhaps give an example but our encoding is a modified base b encoding, where b is the number of symbols in our language, plus a further two to get the extended language. We need b digits and for the language that will be slightly too many for the ordinary decimal digits plus the letters of the English alphabet. I could use a language with a less impoverished alphabet, but the unfamiliarity would probably make the example create more confusion than it would eliminate.

We now have a binary relation between natural numbers, which we would express in English as "... is the encoding of ...". Can we express it in the language of arithmetic? In other words, is this relation arithmetic? In fact it's possible to prove that it is. One approach would be to note that finding the natural number from the encoding is a relatively straightforward computation. It's deterministic but that doesn't prevent us from using the recipe we've already developed for non-deterministic computations. This chapter is already quite proof-heavy though so I won't give the details.

Tarski's theorem

Earlier I discussed the question of whether the encodings of the theorems of a formal system are an arithmetic set. We don't yet have a formal system for arithmetic. That will come soon, but for now we do have a language and an interpretation, so we can ask about the true sentences. By sentence I mean a statement which is unconditionally true or false, since it has no free variables or parameters which need values to be assigned to them. The answer turns out to be no. This is a theorem of Alfred Tarski.

Tarski's theorem is not easy to prove, but the idea behind it is fairly simple. Suppose the set of true sentences is arithmetic, i.e. that we have an expression meaning "... is the encoding of a true statement". We already have, or rather know that there is, an expression meaning "... is the encoding of ...". With a bit of cleverness we can combine these to to create a sentence which asserts that it is false.

A formal system for arithmetic

As mentioned earlier, a formal system consists of a language, a set of axioms, and a set of rules of inference. We have a language for arithmetic but we don't yet have axioms or rules of inference. There are a variety of possible choices which tend to known collectively as Peano arithmetic, after Giuseppe Peano, the first person to introduce such a system.

There are two types axioms and rules of inference, logical axioms and rules of inference and arithmetic axioms and rules of inference.

For the logical part we'll just borrow from the system we've already developed. The only change is that in place of parameters we now have numerical expressions and instead of predicates we now have Boolean expressions. In other words, in place of a first order logic statement like

$$\{[\exists x.(fx)] \supset [\neg(\forall x.\{\neg[(fx) \vee (gx)]\})]\}$$

we have statements like

$$\{[\exists x.(\exists y.\{x = y + y\})] \supset [\neg(\forall x.\{\neg[(\exists y.\{x = y + y\}) \vee (\exists y.\{x = y'\})]\})]\}.$$

We've replaced the generic predicates (fx) and (gx) with the specific expressions $(\exists y.\{x = y + y\})$ and $(\exists y.\{x = y'\})$. The first is the translation

into our language of the statement that x is even and the second is the translation of the statement that x is positive. We could have replaced them with any other Boolean expressions. Indeed that's the point of logic: to determine which statements are universally true simply because of their form, without reference to the meaning of their components.

For example, one of our arithmetic axioms will be

$$[\forall x.\{\forall y.[(x + y)' = (x + y')]\}]$$

One of our rules for quantifiers in first order logic allowed us to take a universal quantifier followed by a variable and an expression, remove the quantifier and variable, and replace all free occurrences of the variable in the expression with a parameter. We can do the same in arithmetic, except now we need to replace the variable with a numerical expression, like $0''$. So from the axiom above we can deduce

$$\{\forall y.[(0'' + y)' = (0'' + y')]\}.$$

The terminology may be unfamiliar but the underlying idea should not be: since we have a statement which is true for all natural numbers x it is true in particular for 2, a.k.a. $0''$.

The rules of inference which deal with quantifiers involve introducing eliminating parameters. As stated above, numerical expressions take the role of parameters. Those expressions could be variables or could be more complicated expressions. To main soundness we need to avoid variable capture, which was discussed earlier in the context of substitution in first order logic, when an expression involving variables is substituted for a variable. Otherwise we would be able to deduce the false statement

$$[\forall z.(\forall y.\{\exists z.[(z + y) = z]\})]$$

from the true statement

$$[\forall x.(\forall y.\{\exists z.[(x + y) = z]\})].$$

Other than the changes described above, for parameters and predicates, the logical structure is just that of first order logic. What's new is the arithmetic axioms and rules of inference.

We'll use the following axioms for arithmetic:

1. $[\exists x.(x = 0)]$
2. $\{\forall x.[\exists y.(y = x')]\}$
3. $\{\forall x.[\neg(x' = 0)]\}$
4. $\{\forall x.[(x + 0) = x]\}$
5. $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
6. $\{\forall x.[(x \cdot 0) = 0]\}$
7. $[\forall x.(\forall y.\{(x \cdot y') = [(x \cdot y) + x]\})]$
8. $\{\forall x.[\forall y.(\forall z.\{[(x + y) = z] \supset [(z - y) = x]\})]\}.$
9. $\{\forall x.[\forall y.(\forall z.\{[(z - y) = x] \supset [(x + y) = z]\})]\}.$
10. $(\forall x.\{\forall y.[(x \leq y) \supset \{\exists w.[(w + x) = y]\}]\})$
11. $(\forall x.\{\forall y.[\{\exists w.[(w + x) = y] \supset (x \leq y)\}]\})$
12. $(\forall x.\{\forall y.[(x \geq y) \supset \{\exists z.[x = (y + z)]\}]\})$
13. $(\forall x.\{\forall y.[\{\exists z.[x = (y + z)] \supset (x \geq y)\}]\})$
14. $[\forall x.(\forall y.\{(x < y) \supset [\neg(y \leq x)]\})]$
15. $[\forall x.(\forall y.\{[\neg(y \leq x)] \supset (x < y)\})]$
16. $[\forall x.(\forall y.\{(x > y) \supset [\neg(y \geq x)]\})]$
17. $[\forall x.(\forall y.\{[\neg(y \geq x)] \supset (x > y)\})]$

Before reading further you might find it useful to translate each of these into words and convince yourself that it's true.

The first axiom is the existence of 0. The second says that every natural number has a successor. The third axiom says that 0 is not the successor of any natural number. The fourth says that 0 is an identity element for addition, or at least is a right identity element. The fact that it's a left identity element as well will be a theorem rather than an axiom The fifth axiom tells us that incrementing a sum is the same as incrementing one of the summands, specifically the second summand. The fact that incrementing the first summand would also work is again a theorem rather than an axiom. The fourth and fifth axioms together are best thought of as a recursive definition of addition. If we know how to add 0 to a number and know how to add the successor of any number to a number then we know how to add any number to it. The sixth axiom tells us that 0 multiplied by anything is still 0. Again, there's a counterpart with the multiplicands in the other order which will be a theorem rather than an axiom. The sixth and seventh axioms are essentially a recursive definition of multiplication. The sixth axiom tells us how to multiply by 0 and the seventh axiom allows us to get, one step at a time, from multiplication by 0 to multiplication by any natural number. The eighth and ninth axioms define subtraction in terms of

addition. The remaining axioms just express the other arithmetic relations in terms of equality.

There are also three arithmetic rules of inference.

1. From a statement of the form $(X = Y)$ we can deduce $(X' = Y')$, and vice versa.
2. $(\forall V < B : P)$ is freely interchangeable with $\forall V.(V < B) \supset P$ and $(\exists V < C : P)$ is freely interchangeable with $\exists V.(V < B) \supset P$, where V is a variable, B a bound, and P a Boolean expression.
3. Suppose V is a variable and P is a Boolean expression. Let Q be P with all free occurrences of V replaced by 0 and let R be P with all free occurrences of V replaced by V'. From Q and $[\forall V.(P \supset R)]$ we can deduce $(\forall V.P)$.

The first part of the first rule is actually redundant. It follows from one of our equality rules in first order logic. More generally, if we have $(X = Y)$ then we can deduce an equality between any two expressions which differ only in that one has X everywhere the other has Y and vice versa. The interesting part of the first rule is therefore the "and vice versa" part. The third rule is just says that bounded quantifiers are a shorthand notation, as explained when they were first introduced.

Induction

The last rule of inference is the formal version of the principle of mathematical induction, which we used once already informally in showing that the Fibonacci numbers form an arithmetic set. My preferred way of thinking about the principle of mathematical induction is as the statement that every non-empty set of natural numbers has a least element.

To see why this minimum principle implies the rule above consider the set of natural numbers which, when substituted for all free occurrences of V in P, yield a false statement. If there are any then there's a least one. It can't be 0 because Q is true. If it's not zero then it's the successor of some natural number. Call that number x. So substituting x' for V in P gives a false statement. But x' was the least number with this property so substituting x would give a true statement. Substituting x' for V in P is the same as substituting x for V in R though and we have $[\forall V.(P \supset R)]$. Substituting x for

V in this, which we are allowed to do by one of logical rules of inference for quantifiers, would give a contradiction, so our assumption that there is an integer which, when substituted into P for V makes the statement false is incorrect. In other words, substituting any value for V gives a true statement. But that's the same as saying that $(\forall V.P)$ is true. So the minimum principle implies the principle of mathematical induction.

The reverse implication works as well. Suppose we have a set of natural numbers with no least element. Let P be the statement that no natural number less than the value represented by V belongs to the set. This is vacuously true when 0 is substituted for V. Suppose it's true for some other value. Then this value does not belong to the given set. If it did then it would be the least element of the set because the statement P tells us that no smaller number belongs to the set. Since there is no least element this can't happen so the value V is not in the set. But then all numbers smaller than the value V' are not in the set so from P we can deduce P with V replaced by V', i.e. the statement we previously called R. So we now have Q and $[\forall V.(P \supset R)]$ and therefore, by the principle of mathematical induction, $(\forall V.P)$. But P is the statement that no number less than V belongs to the set. This holds with V replaced by any numerical expression, including x', where x is a variable. So no number less than $x';o$ belongs to the set and in particular x does not belong to the set. This holds for all natural numbers x so no natural number belongs to the set, which must therefore be empty. We've just seen that a set of natural numbers with no least element is necessarily empty. An equivalent way to say this is that every non-empty set of natural numbers has a least element, which is our minimum principle.

The proof above is an informal one. Indeed it can't help but be informal. Our language for arithmetic has no notation for sets of natural numbers. We've seen how to express particular sets in this language but that's not sufficient for the minimum principle, which is a statement about all sets of natural numbers. So there's no way within Peano arithmetic to state the minimum principle, let alone prove its equivalence to the principle of mathematical induction. Once we have a language which includes sets, like the one we'll introduce in the next chapter, we can give a formal statement of the minimum principle.

Formal proofs

It is possible, though not very pleasant, to produce formal proofs in Peano arithmetic.

Consider, for example, the following proof of the fact that $2 + 2 = 4$, which in the language we're using is written as $[(0'' + 0'') = 0'''']$.

1. $[\exists x.(x = 0)]$
2. $[\exists y.(y = 0)]$
3. $\{\forall x.[\exists y.(y = x')]\}$
4. $\{[\exists x.(x = 0)] \supset [\exists y.(y = 0')]\}$
5. $[\exists y.(y = 0')]$
6. $[\exists x.(x = 0')]$
7. $\{[\exists x.(x = 0')] \supset [\exists y.(y = 0'')]\}$
8. $[\exists y.(y = 0'')]$
9. $[\exists x.(x = 0'')]$
10. $\{\forall x.[(x + 0) = x]\}$
11. $\{[\exists x.(x = 0'')] \supset [(0'' + 0) = 0'']\}$
12. $[(0'' + 0) = 0'']$
13. $[(0'' + 0)' = 0''']$
14. $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
15. $([\exists x.(x = 0'')] \supset \{\forall y.[(0'' + y') = (0'' + y)']\})$
16. $\{\forall y.[(0'' + y') = (0'' + y)']\}$
17. $\{[\exists y.(y = 0)] \supset [(0'' + 0') = (0'' + 0)']\}$
18. $[(0'' + 0') = (0'' + 0)']$
19. $[(0'' + 0') = 0''']$
20. $[(0'' + 0')' = 0'''']$
21. $\{[\exists y.(y = 0')] \supset [(0'' + 0'') = (0'' + 0')']\}$
22. $[(0'' + 0'') = (0'' + 0')']$
23. $[(0'' + 0'') = 0'''']$

Here 1 is an axiom, and 2 is the result of substitution on 1. 3 is an axiom, 4 is obtained from 3 by one of our quantifier rules, 5 is modus ponens applied to 1 and 4, and 6 is substitution on 6. 7 is a quantifier rule applied to 3, 8 is modus ponens applied to 6 and 9, and 9 is substitution on 8. 10 is an axiom, 11 is a quantifier rule applied to 10, and 12 is modus ponens on 9 and 11. 13 is derived from 12 by one of our arithmetic rules of inference. 14 is an axiom, 15 is derive from it by a quantifier rule, and 16 is modus ponens

on 9 and 15. 17 is a quantifier rule applied to 16 and 18 is modus ponens on 2 and 17. 19 is derived from 13 and 18 by one of our rules of inference for equality. 20 comes from an arithmetic rule of inference on 19. 21 is a quantifier rule applied to 16 and 22 is modus ponens on 5 and 21. Finally, 23 is one of the rules of equality applied to 20 and 22.

Statements more complicated than $2 + 2 = 4$ have correspondingly longer proofs. The following is a proof of

$$[\forall x.(\forall y.\{\exists z.[z = (x + y)]\})],$$

i.e. the fact that the sum of any two natural numbers is a natural number.

1. $[\exists x.(x = 0)]$
2. $(0 = 0)$
3. $\{[\exists x.(x = 0)] \wedge (0 = 0)\}$
4. $[\exists x.(x = x)]$
5. $[\exists y.(y = y)]$
6. $[\exists z.(z = z)]$
7. $(z = z)$
8. $\{[\exists z.(z = z)] \wedge (z = z)\}$
9. $[\exists z.(z = x)]$
10. $[\exists z.(z = w)]$
11. $[\exists x.(x = w)]$
12. $\{\forall x.[\exists y.(y = x')]\}$
13. $\{[\exists x.(x = w)] \supset [\exists y.(y = w')]\}$
14. $[\exists y.(y = w')]$
15. $[\exists z.(z = w')]$
16. $\{\forall x.[(x + 0) = x]\}$
17. $\{[\exists x.(x = x)] \supset [(x + 0) = x]\}$
18. $[(x + 0) = x]$
19. $(\exists z.\{[z = (x + 0)]\})$
20. . $(\exists z.\{[z = (x + y)]\})$
21. . $([\exists z.(z = w)] \wedge \{[w = (x + y)]\})$
22. . $[w = (x + y)]$
23. . $[w' = (x + y)']$
24. . $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
25. . $([\exists x.(x = x)] \supset \{\forall y.[(x + y') = (x + y)']\})$
26. . $\{\forall y.[(x + y') = (x + y)']\}$

27. . $\{[\exists y.(y = y)] \supset [(x + y') = (x + y)']\}$
28. . $[(x + y') = (x + y)']$
29. . $[(x + y)' = (x + y')]$
30. . $[w' = (x + y')]$
31. . $\{[\exists z.(z = w')] \wedge [w' = (x + y')]\}$
32. . $\{\exists z.[z = (x + y')]\}$
33. $\{\exists z.[z = (x + y)]\} \supset \{\exists z.[z = (x + y')]\})$
34. $\{\exists z.[z = (x + y)]\}$
35. $\{\neg[\exists y.(y = y)]\} \vee \{\exists z.[z = (x + y)]\}$
36. $[\exists y.(y = y)] \supset \{\exists z.[z = (x + y)]\}$
37. $(\forall y.\{\exists z.[z = (x + y)]\})$
38. $[\{\neg[\exists x.(x = x)]\} \vee (\forall y.\{\exists z.[z = (x + y)]\}])$
39. $\{[\exists x.(x = x)] \supset (\forall y.\{\exists z.[z = (x + y)]\}])\}$
40. $[\forall x.(\forall y.\{\exists z.[z = (x + y)]\}])]$

This one is somewhat more complicated than the previous one. A hypothesis is introduced at 20 and then discharged at 33. This, together with 19, allows us to use induction at 34.

Had we used a logic with existential presuppositions the proof could have been reduced to four lines. That's largely because such a logic assumes that any expression we can write down refers to something in the domain, so the fact that we have a notation for addition already essentially assumes that the sum of any two natural numbers is a natural number. That's very convenient, but unfortunately the same would apply to subtraction. If we introduce a notation for subtraction into a system based on such a logic then we can give a four line proof of

$$[\forall x.(\forall y.\{\exists z.[z = (x - y)]\}])],$$

but unfortunately this statement is false. The difference of natural numbers need not be a natural number. Formal systems for elementary arithmetic based on first order logic with existential suppositions avoid this problem by not having a notation for subtraction. But then there are important facts about elementary arithmetic which they can't express. For example, the correct version of the statement above is

$$(\forall x.\{\forall y.[(x \geq y) \supset \{\exists z.[z = (x - y)]\}]\}),$$

which says that $x - y$ is a natural number if $x \geq y$. This is a theorem in our system but can't even be stated in a system with existential suppositions.

As we saw earlier, it's possible to give a fairly concise statement within the language of Peano arithmetic of the fact that there are infinitely primes. It's possible to give a formal proof as well, but it's hardly an enjoyable exercise. Logicians, though, are generally much more interesting in figuring out what can or can't be proved within a formal system than with actually supplying proofs. In other words, they tend to live in the world of semiformal proofs rather than formal proofs.

Gödel's theorem

Now that we have a formal system for arithmetic we can do what we discussed earlier and show that the set of theorems is arithmetic. If we've already done this for first order logic, which I haven't but I did mention that this can be done, it's not even particularly difficult. The axioms and the first rule of inference present no problems. The second rule of inference is more complicated but recognising instances of this rule is essentially a matter of pattern matching, and we've seen how to do pattern matching within arithmetic. Not every theorem is a sentence, but recognising which ones are is also a pattern matching problem and so the set of sentences which are theorems is also arithmetic.

Tarski's theorem, that true sentences are not an arithmetic set, is interesting in itself, but it's particularly interesting in combination with the observation that the set of provable sentences is arithmetic. This leads to the conclusion, first proved by Kurt Gödel, that these two sets are not the same, i.e. that there must either be a sentence which is true but cannot be proved, or a sentence which is false but can be proved!

The formulation above is somewhat sloppy. For simplicity I've referred to sets of sentences as being arithmetic or not, but arithmeticity is a property which applies to sets of natural numbers rather than sentences. What I really mean is that the sets of encodings of those sentences are arithmetic or not. But the set of encodings determines the set of sentences uniquely, so if the sets of encodings are different then so are the sets of sentences. So our final conclusion doesn't reference the encodings.

If this were restricted to the formal system above then it wouldn't have much interest. After all, the precise formal system above appears in these notes and nowhere else, so finding out that it's incapable of proving all true

statements in arithmetic isn't terribly interesting. Perhaps I just need to add a new axiom or strengthen one of the rules of inference? The argument that led us to this point is quite a general one, though. Tarski's theorem didn't depend on the choice of formal system at all. The proof that theorems are an arithmetic set didn't use any properties of this particular formal system. I haven't defined the notion of a formal system precisely, but under any definition in which we can mechanically check whether proofs are valid we will get the same conclusion. So Gödel's theorem is not a theorem about a particular formal system for arithmetic but rather a theorem about all possible formal systems for arithmetic.

Rosser's theorem

Gödel did not prove his theorem as a consequence of Tarski's theorem, which wasn't proved until a couple of years later. In fact Tarski's proof of his theorem was inspired by Gödel's proof of his.

One way to think about Gödel's theorem is that it says a formal system for arithmetic can be sound or semantically complete, but not both. Equivalently, every sound system is semantically incomplete. A sound system which is semantically complete is syntactically complete, so an alternate approach would be to prove first that every sound system is syntactically incomplete, and then derive semantic incompleteness as a result. This is roughly what Gödel tried to do. What he got wasn't quite syntactic incompleteness but was sufficient to prove semantic incompleteness. A few years after Tarski proved his theorem Barkley Rosser succeeded in constructing a proof along the lines which Gödel had initially attempted, showing that a system for arithmetic can be consistent or syntactically complete, but not both. The main interest of this result is that the interpretation of the system now plays only a very minor role, since consistency and syntactic completeness require from the interpretation only a concept of negation. The interpretation is still lurking in one other place though. For any of these theorems to apply we need a certain level of descriptive completeness of our system; it has to be capable of describing enough of elementary arithmetic to be able to carry out an encoding in such a way that concatenation is representable within the system.

One final historical note is that Gödel's first goal was not to prove any of

the theorems mentioned above but rather to prove that Peano arithmetic is inconsistent! We now know that it's at most consistent or complete but we don't know which. Most logicians and mathematicians assume it's consistent and therefore not complete, which is why Gödel's theorem is referred to as Gödel's first incompleteness theorem–there is also a second theorem–rather than Gödel's first inconsistency theorem. We could be wrong though. Gödel himself, at least at some point, must have thought that we were or he wouldn't have set out to prove inconsistency. Gödel was very clever so even though he didn't actually succeed in proving Peano arithmetic's inconsistency we should perhaps be more cautious in assuming its consistency.

Set theory

Elementary arithmetic is arithmetic without sets, or, more precisely, arithmetic with no notation for sets. We can refer to sets indirectly, by means of the expressions which could be used to define them, but we can't name a set and we can't quantify over sets. This prevents us expressing concepts like our minimum principle, that every non-empty set of natural numbers has a least member.

We now move on to set theory. Set theory, like first order logic, is generally used as a base for other, more interesting theories. Just as in first order logic we didn't enquire too closely into the meanings of variables and predicates, in pure set theory we mostly avoid the question "sets of what?" Sets are sets of members. For now that's all we need to know.

Set theory is weird. To be more precise, it's weird in two ways. One is that various statements each of which individually seem to be intuitively obvious turn out to be logically inconsistent when combined. This means that any choice of axioms for set theory will necessarily have some unexpected consequences. The other way that it's weird is that the particular set of axioms which the mathematical world has converged on has somewhat more unexpected consequences than strictly necessary.

A language for set theory

As usual, we'll start with a language, and that language will be based on first order logic. This language is described by the following phrase structure grammar.

```
statement : bool_exp
bool_exp | "[" "¬" bool_exp "]" | "[" bool_exp b_operator bool_exp "]"
         | "[" quantifier variable "." bool_exp "]"
         | "[" quantifier variable "∈" set_exp : bool_exp "]"
         | "[" variable relation variable "]"
b_operator : "∧" | "∨" | "⊃"
quantifier : "∀" | "∃"
relation : "∈" | "=" | "⊆"
set_exp : "∅" | variable | "[" "∩" set_exp "]" | "[" "∪" set_exp "]"
       | "[" set_exp "∩" set_exp "]" | "[" set_exp "∪" set_exp "]"
       | "[" set_exp "∖" set_exp "]" | "[" set_exp "×" set_exp "]"
        | "[" "P" set_exp "]" | "{" list "}" | "(" list ")"
        | "{" variable "∈" set_exp : bool_exp "}"
list : | sequence
sequence : set_exp | set_exp "," sequence
variable : letter | variable "!"
letter : "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E"
      | "F" | "G" | "H" | "I" | "J" | "R" | "S" | "T" | "U" | "V"
```

Some of these symbols and rules are familiar from earlier chapters while others are new, or are used in new ways. The following remarks refer to the intended interpretation of the language and are not strictly part of the language.

The symbols (,), {, and } are no longer used for grouping expressions but have special meanings. Only [and] are used for grouping expressions as in previous chapters. { and } are used in two different ways of constructing sets. We write $\{x, y, z\}$ for the set whose only members are x, y and z, for example, and $\{x \in A : [\neg[x \in B]]\}$ for the set of x which are members of A but not of B. The $\in$ sign denotes set membership. (and) are used in two ways. One is to construct lists. (x, y, z) is the list whose first element is x, whose second element is y and whose third element is z. This is unlike $\{x, y, z\}$, where x happens to have been written first, y happens to have been

written second, and z happens to have been written third, but the ordering is not significant. $\{x, y, z\}$ is the same set as $\{y, z, x\}$ but (x, y, z) is not the same list as (y, z, x) unless x, y and z are all equal.

In addition to the old notation for quantifiers there is a new notation. We have expressions like $[\forall x \in A : [[x \in B] \vee [x \in C]]]$. This is to understood as a shorthand for $[\forall x.[[x \in A] \supset [[x \in B] \vee [x \in C]]]]$. This, and the corresponding notation for $\exists$, are convenient because we often want to state that all members of a set have some property or that some member has that property. Such quantifiers are called bounded quantifiers. We met similar bounded quantifiers in constructive elementary arithmetic.

It's possible to do first order logic with bounded quantifiers as if they were ordinary quantifiers, provided all quantifiers are over the same set, at least with the version of first order logic we're using, one without existential presuppositions. In a logic with existential presuppositions we would need to add the additional requirement that the set is known in advance to be non-empty. It's easy, but dangerous, to forget the non-emptiness requirement.

We have a new relation $\subseteq$ as well. This is the subset relation. Note that this is not necessarily a proper subset. Each set is a subset of itself. The distinction between $\in$ and $\subseteq$ was a source of confusion in the early development of set theory. It's still often a source of confusion for students. $A \in B$ means that A is a member of B while $A \subseteq B$ means that every member of A is a member of B.

$\emptyset$ is the empty set, i.e. the set with no members. There are two versions of the intersection and union operators. The unary operators $\bigcap$ and $\bigcup$ indicate intersection and union, respectively, and are not preceded by a set expression. $\bigcup$ takes a set and gives you its union, i.e. the set whose members are the members of its members. In other words, $[x \in [\bigcup A]]$ if and only if there is some B such that x is a member of B and B is a member A. The most common case of unions is one where the set of sets has two member so we have a special notation for this. We write $[B \cup C]$ to mean $[\bigcup\{B, C\}]$, i.e. the set of all members of B which are in C. Similar remarks apply to the intersection. If A is a non-empty set then $[x \in [\bigcap A]]$ if and only if x is a member of B for every B in A. The restriction to non-empty sets will be explained later. Again we have a special notation for the intersection of a pair of sets. $[B \cap C]$ means $[\bigcap\{B, C\}]$. If you have trouble visually distinguishing $\bigcap$ from $\cap$, or $\bigcup$ from $\cup$, or if you're using a screen reader

which reads them the same, you needn't worry too much. The grammar is designed in such a way that they can always be distinguished by context, specifically by whether or not they are preceded by a set expression.

Our other two set operations are $\setminus$ for the relative complement and $\times$ for the Cartesian product. $[A \setminus B]$ is the set of members of A which are not in B, i.e. $\{x \in A : [\neg[x \in B]]\}$. $[A \times B]$ is the set of pairs (x, y) where x is an member of A and y is a member of B. $[PA]$ is the power set of A, i.e. the set of all subsets of A, while $[LA]$ is the set of lists all of whose elements are members of A.

You may notice that I've dropped practice of using separate symbols, outside the language, for expressions of various types. Occasionally it will be convenient to introduce an ad hoc notation but I'll mostly do what I've done above, and use a particular variable to stand for any variable, or sometimes any set expression. Hopefully this will not cause confusion.

Simple set theory

We'll start with a subset of set theory which is almost sufficient for almost all of mathematics and computer science. As we did with elementary arithmetic we'll borrow first order logic. We won't borrow elementary arithmetic itself. In particular we will not assume that numbers exist. You may have noticed that the language I've introduced has no notation for them.

Axioms (informal version)

Our axioms are

- Extensionality: Suppose A and B are sets and every member of A is a member of B and vice versa then $A = B$.
- Elementary sets: There is a set $\emptyset$ such that for all x we have $[\neg[x \in \emptyset]]$. For all x we have a set $\{x\}$, of which x is a member and there are no other members. Similarly, for all x and y we have a set $\{x, y\}$ such that x and y are members and there are no other members.
- Separation: For every variable x, set A and Boolean expression θ the set $\{x \in A : \theta\}$, whose members are those members of A for which θ

is true, exists.

- Power set: For any set A the power set $[PA]$ exists. $[B \in [PA]]$ if and only if every member of B is a member of A.
- Union: For every set A the set $[\bigcup A]$ exists. This is the set of all members of members of A.

These are informal statements of the axioms. Their formal equivalents will be given shortly, but they are hard to read if you don't know what they're trying to express. Before doing that, there are a few things to notice about these axioms.

Discussion

The Axiom of Extensionality tells us that sets are characterised purely by their members. This is something which often causes confusion. We have many ways of describing sets, but the set is not its description. In terms of our language, there could be multiple set expressions which describe the same set. There could also be no set expression which describes a particular set.

The Axiom of Elementary Sets has some redundancy. The only parts we really need are are the existence of some set and the fact that for all x and y there is a set of which x and y are members. If A is such a set then $\{w \in A : [[w = x] \vee [w = y]]\}$ is a set of which they are the only members. There can only be one such set, by the Axiom of Extensionality. This is the set we called $\{x, y\}$. We don't really need to state the existence of $\{x\}$ separately. It's the same as $\{x, x\}$. Also, if A is a set $\{x \in A : [\neg[x = x]]\}$ is a set, by the Axiom of Separation, and has no members. By the Axiom of Extensionality there can only be one such set. This is the set we called $\emptyset$. So if there are any sets at all then there is an empty set. What about sets with more than two members? We can show that those exist using this axiom together with the Axiom of Union. $\{x, y, z\}$, for example, is $[\bigcup\{\{x, y\}, \{y, z\}\}]$.

The Axiom of Separation is not an axiom. Instead it's what's called an axiom schema, i.e. a common pattern for a family, indeed an infinite family, of axioms, one for each choice of variable, set and expression. It would have been better to make this into a rule of inference rather than an axiom but for historical reasons it is called an axiom. Note that the axiom doesn't

allow us to use an expression to construct a set of everything which makes that expression true, only to construct a set of those members of a given set which make the expression true. In other words, it carves out a subset from a set which is already known to exist. It can't create sets from nothing. If you've been reading carefully you'll have realised that there's something wrong with my informal description of the Axiom of Separation. I referred to members for which a statement is true. Truth has no place in a formal system. The formal version of the axiom, or rather axiom schema, does not refer to the concept of truth.

The Axiom of Separation is useful for constructing particular subsets but it doesn't assure us that the subsets of a given set form a set. For that we need the Power Set Axiom. I haven't actually said what a subset is but you can probably guess. A is a subset of B, written $[A \subseteq B]$ if every member of A is a member of B. As with various other axioms, instead of assuming the existence of the power set itself we could just assume the existence of some set such that all the subsets of A are members of it and then use the Axiom of Separation to remove any members which aren't subsets of A.

The Axiom of Union is used to create unions. That's straightforward enough. As with most other axioms we could just assume the existence of some set such that every member of every member of A is a member of it, and then use the Axiom of Separation to remove any other members.

What may seem odd is the absence of any Axiom of Intersection. We don't need one. We can define $[\bigcap A]$ as

$$\{x \in [\bigcup A] : [\forall B \in A : [x \in B]]\}.$$

In other words, x belongs to $[\bigcap A]$ if and only if it is a member of some member of A which is also a member of all members of A. If A is a non-empty set then the condition that it's a member of all members of A implies that it's a member of some member of A. We couldn't just have defined it as

$$\{x.[\forall B \in A : [x \in B]]\}$$

though. The Axiom of Separation requires us to restrict x to a set. If A is not a non-empty set then the definition above would give $[[\bigcap A] = \emptyset]$. This would have some unfortunate consequences. For example, it would not be true that $[A \subseteq B]$ implies $[[\bigcap B] \subseteq [\bigcap A]]$. For this reason it's better just

not to define $[\bigcap A]$ unless A is a non-empty subset. This okay because our version of first order logic allows expressions which don't refer to anything.

The axioms above mix assumptions about the existence of sets with notations for them. Strictly speaking the axioms are just the part which assert the existence of a set. One important point to understand is that having a notation for something doesn't mean it exists. Mathematics is full of notations for things which don't exist, like $1/0$. The axioms of set theory are arranged in such a way that everything we have a notation for will in fact exist, but it exists as a consequence of the axioms, not just because we happen to have included it in our language. We have a number of notations for which there are no axioms. The intersection symbol, which we just considered, is such a notation. Others are the notations for set differences, lists and Cartesian products. We'll need to give definitions for those, just as we did for the intersection.

Axioms (formal version)

Here are the formal versions of the axioms.

- Extensionality:

  $$[\forall A.[\forall B.[[\exists x.[x \in A]] \supset [[\forall y.[[[y \in A] \supset [y \in B]] \wedge [[y \in B] \supset [y \in A]]]] \supset [A = B]]]]]$$

- Elementary Sets:

  $$[\forall x.[\neg[x \in \emptyset]]]$$

  and

  $$[\forall x.[\forall y.[\exists A.[[x \in A] \wedge [y \in A]]]]]$$

- Separation:

  $$[\exists B.[\forall x.[[[x \in B] \supset [[x \in A] \wedge \theta]] \wedge [[[x \in A] \wedge \theta] \supset [x \in B]]]]]$$

  Here x can be replaced by any variable, A by any set expression and θ by any Boolean expression in which B has no free occurrences.

- Power Set:

  $$[\forall A.[\exists B.[\forall C.[[\forall x.[[x \in C] \supset [x \in A]]] \supset [C \in B]]]]]$$

- Union:

$$[\forall A.[\exists B.[\forall C \in A : [\forall x \in C : [x \in B]]]]]$$

These aren't quite the axioms as they appeared initially. Instead I have incorporated some of the observations from the discussion section to shorten the axioms. For example, the formal version of the Axiom of Elementary Sets just says that $\emptyset$ is the empty set and that for all x and y there is a set with both x and y as members, not that there is a set with only those members, and doesn't say anything about sets with only one member. The axiom above is therefore also known as Axiom of Pairing.

Non-sets

There is no set of all sets. This follows directly from the axioms. Suppose there were a set A such that every set is a member of A. By the Axiom of Separation then we can form the set

$$B = \{C \in A : [\neg[C \in C]]\}.$$

In other words C is the set of all sets which are not members of themselves. Is B a member of B? If not then B is a set which is not a member of itself, but then by the definition of B it is a member of B. Similarly, if B is a member of B then it doesn't satisfy the definition of B and so isn't a member of B. So the assumption that there is a set of all sets leads to a contradiction.

A is a set if and only if it's the empty set or has at least one member. In other words, sets are characterised by the Boolean expression

$$[[A = \emptyset] \vee [\exists x.[x \in A]]].$$

In the Axiom of Separation we weren't allowed to define sets with just a Boolean expression; we needed a Boolean expression and some other set. In other words, expressions were used to define subsets of a given set, not to define sets directly. Now we can see why. If we could just use expressions to define sets then we could define the set of all sets as

$$\{A : [[A = \emptyset] \vee [\exists x.[x \in A]]]\}.$$

But we've just seen that this set can't exist, so an axiom which allowed us to define it would necessarily be unsound.

More generally, there is no such thing as the complement of a set. The complement of the empty set would be the set of all sets, and we've already seen that that doesn't exist. The same holds for any set though. Suppose the complement of A existed, i.e. that there was a set C such that every member of A is not a member of C, and vice versa. By the Axiom of Pairing there is then a set B with both A and C as members. By the Axiom of Union there's then a set D such that every member of every member of B is a member of D, and in particular every member of A or C is a member of D. Everything is either in A or C though and therefore in D. I've been deliberately rather vague about whether our language is meant to include objects which are not sets, a point we'll need to return to later, but in either case we can use the Axiom of Separation to define

$$E = \{x \in D : [[x = \emptyset] \wedge [\exists y.[y \in x]]]\}.$$

This E is the set of those members of D which are sets, and so is the set of all sets, which we've already seen doesn't exist. So there can be no such set B.

Relative complements are meaningful though. $[A \setminus B]$ is easily defined as

$$[A \setminus B] = \{x \in A : [\neg[x \in B]]\}.$$

In some contexts we're only concerned with subsets of one given set. We might, for example, be discussing subsets of the natural numbers, and only subsets of the natural numbers. In such a case it's common to drop the word "relative" and just say "complement". This is just shorthand though and the set described in this way is still a relative complement.

In the discussion of first order logic I described a class of interpretations where the variables were to be understood as members of a set and mentioned that these were not the only interpretations. We can now see why. We're applying first order logic to set theory, and the variables are allowed to range over all sets, but there is no set of all sets, so this cannot be an interpretation of the type described earlier.

The non-existence of the set of all sets has some other awkward consequences. We've already met one of them. It's what prevented us from defining $[\bigcap \emptyset]$ to be the union of all sets, which might otherwise have seemed like a way to extend the unary intersection operator to all sets while maintaining the property that if $[A \subseteq B]$ then $[[\bigcap B] \subseteq [\bigcap A]]$.

Set operations and Boolean operations

We can derive a number of set theory identities from zeroeth order logic identities. The basis for this is the following facts.

- $[A \subseteq B]$ if and only if $[[x \in A] \supset [x \in B]]$. Indeed this is just the definition of the $\subseteq$ relation.
- If $[A \subseteq B]$ and $[B \subseteq A]$ then $[A = B]$. This is a consequence of Extensionality.
- $[[x \in [A \cap B]]$ if and only if $[[x \in A] \wedge [x \in B]]$. This is more or less the definition of the $\cap$ operator.
- $[[x \in [A \cup B]]$ if and only if $[[x \in A] \vee [x \in B]]$. This is more or less the definition of the $\cup$ operator.
- $[[x \in [A \setminus B]]$ if and only if $[\neg[[x \in A] \supset [x \in B]]]$. This is more or less the definition of the $\setminus$ operator.

So the three set operators $\cap$, $\cup$ and $\setminus$ are expressible in terms of the four Boolean operators $\wedge$, $\vee$, $\neg$, and $\supset$. $\cap$ corresponds to $\wedge$ and $\cup$ corresponds to $\vee$, which is fairly easily to remember. $\setminus$ corresponds to a particular combination of $\neg$ and $\supset$, but no set operator corresponds to $\neg$ or $\supset$ individually. In some sense the complement operator, if there were one, would correspond to $\neg$.

As an example, consider the associativity of the union operation, i.e. the identity $[[[A \cup B] \cup C] = [A \cup [B \cup C]]]$. $[[p \vee [q \vee r]] \supset [[p \vee q] \vee r]]$ is a tautology in zeroeth order logic. Substituting $[x \in A]$ for p, $[x \in B]$ for q, and $[x \in C]$ for r gives

$$[[[x \in A] \vee [[x \in B] \vee [x \in C]]] \supset [[[x \in A] \vee [x \in B]] \vee [x \in C]]].$$

We can replace $[[x \in B] \vee [x \in C]]$ with $[x \in [B \cup C]]$ and $[[x \in A] \vee [x \in B]]$ with $[x \in [A \cup B]]$, so

$$[[[x \in A] \vee [x \in [B \cup C]]] \supset [[[x \in [A \cup B]]] \vee [x \in C]]].$$

Then we can replace $[[x \in A] \vee [x \in [B \cup C]]]$ by $[x \in [A \cup [B \cup C]]]$ and $[[[x \in [A \cup B]]] \vee [x \in C]]$ by $[x \in [[A \cup B] \cup C]]$, so

$$[[x \in [A \cup [B \cup C]]] \supset [x \in [[A \cup B] \cup C]]]$$

and hence

$$[[A \cup [B \cup C]] \subseteq [[A \cup B] \cup C]].$$

Similarly, $[[[p \vee q] \vee r] \supset [p \vee [q \vee r]]]$ is a tautology so

$$[[[A \cup B] \cup C] \subseteq [A \cup [B \cup C]]].$$

Combining that with the inclusion already obtained gives

$$[[[A \cup B] \cup C] = [A \cup [B \cup C]]].$$

The following facts about sets can similarly be proved using tautologies borrowed from zeroeth order logic.

- $[[A \cap B] \subseteq A]$
- $[[A \cap B] \subseteq B]$
- $[A \subseteq [A \cup B]]$
- $[B \subseteq [A \cup B]]$
- $[[A \setminus B] \subseteq A]$
- $[[A \cap A] = A]$
- $[[A \cup A] = A]$
- $[[A \cap B] = [B \cap A]]$
- $[[A \cup B] = [B \cup A]]$
- $[[[A \cap B] \cap C] = [A \cap [B \cap C]]]$
- $[[[A \cup B] \cup C] = [A \cup [B \cup C]]]$
- $[[[A \cap [B \cup C]] = [[A \cup C] \cap [B \cup C]]]$
- $[[[A \cup [B \cap C]] = [[A \cap C] \cup [B \cap C]]]$
- $[[A \cap [A \cup B]] = A]$
- $[[A \cup [A \cap B]] = A]$
- $[[A \setminus [A \setminus B]] = [A \cap B]]$
- $[[C \setminus [A \cap B]] = [[C \setminus B] \cup [C \setminus A]]]$

- $[[C \setminus [A \cup B]] = [[C \setminus B] \cap [C \setminus A]]]$
- $[[A \setminus [B \setminus C]] = [[A \cap C] \cup [B \setminus C]$
- $[[[A \setminus B] \setminus C] = [A \setminus [B \cup C]]]$
- $[[[A \setminus B] \cap C] = [A \cap [C \setminus B]]]$

Finite sets

There are a few different ways to define finiteness of sets. The method below is due to Tarski. It requires some preliminary definitions. To improve readability I'll start being less strict about the bracketing of expressions.

Definitions

A minimal member of a set A is a set C such that $C \in A$ and if $B \in A$ and $B \subseteq C$ then $B = C$. In other words, C is a member of A and no proper subset of C is a member of A. A maximal member of a set A is a set B such that $B \in A$ and if $C \in A$ and $B \subseteq C$ then $B = C$. In other words, B is a member of A and is not a proper subset of any member of A. Sets can have more than one minimal or maximal member. Suppose $x \neq y$. Then $\{x\}$ and $\{y\}$ are both minimal and maximal members of the set $\{\{x\}, \{y\}\}$.

A set E is said to be finite if every non-empty set of subsets of E has both a minimal and a maximal member. It is said to be infinite if it is not finite.

Your intuitive notion of finiteness probably involves associating a natural number to each finite set, the number of members in the set. This assignment probably has the property that if A is a proper subset of a finite set B then A is also finite and the number of members of A is less than the number of members of B. Assuming for a moment that your intuition is correct we can see that every set which is finite according to your intuition is also finite according to the definition above. Let E be finite according to your intuition and let A be a non-empty set of subsets of E. The set of numbers of members of members of A is a non-empty set of natural numbers and therefore has a least member. This number is the number of members of some member of A. Call that member C. If $B \in A$ and $B \subseteq C$ then B can't have fewer members than C because the number of members of C is the least possible number of members for a member of A. It is therefore not

a proper subset, so $B = C$. In other words B is a minimal member of A. The argument to show that A has a maximal member is very similar. We look for a member C of A with the largest possible number of members. Of course subsets of the natural numbers don't have to have a largest member but in this case we only need to consider subsets of E and they have at most as many members as E has, so there is an upper bound on this set and therefore there is a largest member.

The intuitive notion of finiteness considered above can't be turned into a definition. It requires a number of facts about sets which we haven't yet proved. In particular it requires one fact about sets, that proper subsets have a strictly smaller number of members than the whole set, which is only true of finite sets, so even if we had the notions of integers and cardinalities of sets and all their properties we would still be left with a circular definition. That's why we need a definition like the one above.

The argument above just showed that sets you would intuitively regard as finite are finite according to the definition. It didn't show that sets you would intuitively regard as infinite are infinite according to the definition. Part of your intuition for infinite sets is probably that they have arbitrarily large finite subsets. In other words, if E is infinite according to your intuition then we can find a subset D_m with m members for each natural number m. Let B_m be the union of D_k for each $k \leq m$ and let A be the set of all B_m's. If E were finite according to the definition then some member of A would be maximal. It would have to be B_m for value of m because those are the only members of A. Let n be the number of members of B_m. $m \leq n$ because B_m has m members and is a subset of C. Let $C = B_{n+1}$. Then $C \in A$. Also, $B_m \subseteq C$ because B_m is the union of D_k for $k \leq m$ and C is the union of D_k for $k \leq n + 1$ and $m < n + 1$. Since we've assumed B_m is maximal it follows that $B_m = C$. But B_m has n members and D_{n+1}, which is a subset of C, has $n + 1$ members, so we have a subset with more members than the whole set, which is impossible. So our assumption that E is finite according to the definition is untenable. In other words, every set which is infinite according to your intuition is also infinite according the definition.

As with the previous argument this one can't really be formalised because the intuitive notion of finiteness is vague and, if pushed too far, circular. That's why we need a formal definition, which is necessarily somewhat unintuitive. There are two standard choices. One is the definition above,

due to Tarski. The other choice, due to Dedekind, is to define infinite sets to be those which have a proper subsets with the same number of members as the whole set, and then to define infinite to mean not finite. Tarski's definition is better adapted to proving that finite sets have the properties you would expect them to have, e.g. that every subset of a finite set is finite.

Our definition says that E is finite if every member of $PPE \setminus \emptyset$ has a maximal member and that every member of $PPE \setminus \emptyset$ has a minimal member. In fact either of these conditions implies the other. Suppose, for example, that every $A \in PPE \setminus \emptyset$ has a maximal member and that $B \in PPE \setminus \emptyset$. Since every $A \in PPE \setminus \emptyset$ has a maximal member it follows that

$$A = \{C \in PE : \exists D \in B : C = E \setminus D\}$$

has a maximal member. This A is just the set of relative complements of the members of B. If C is a maximal member of A then $E \setminus C$ is a minimal member of B, so B has a minimal member. The argument above shows that if every member of $PPE \setminus \emptyset$ has a maximal member then every member of $PPE \setminus \emptyset$ has a minimal member. The same argument, but with the words minimal and maximal switched, shows that if every member of $PPE \setminus \emptyset$ has a minimal member then every member of $PPE \setminus \emptyset$ has a maximal member. So in order to prove that a set is finite it suffices to prove one condition or the other; we don't have to prove both.

Elementary properties of finite sets.

The empty set is finite. Indeed, $P\emptyset = \{\emptyset\}$, $PP\emptyset = \{\emptyset, \{\emptyset\}\}$ and $PP\emptyset \setminus \emptyset = \{\{\emptyset\}\}$. It's only member, $\{\emptyset\}$, has $\emptyset$ as both a minimal and maximal member.

A set with only one member is finite. Let $A = \{a\}$. Then $PA = \{\emptyset, A\}$ and $PPA = \{\emptyset, \{\emptyset\}, \{A\}, \{\emptyset, A\}\}$. The non-empty members are $\{\emptyset\}$, $\{A\}$ and $\{\emptyset, A\}\}$. The first of these has $\emptyset$ as a minimal and maximal member. The second has A as a minimal and maximal member. The third has $\emptyset$ as a minimal member and A as a maximal member.

We could do a similar case by case analysis to show that sets with two members are finite but it's better just to prove that the union of two finite sets is finite and use the fact that a set with two members is the union of two sets with one member.

Before considering unions we consider subsets, intersections and relative complements, all of which are easier. We start with subsets. Suppose E is finite and $D \subseteq E$. If $A \in PPD \setminus \emptyset$ then $A \in PPE \setminus \emptyset$. E is finite so A has a minimal member. So every non-empty set of subsets of D has a minimal member. We've already seen that if every non-empty set of subsets of a set has a minimal member then that set is finite. So D is finite.

As an easy consequence if A or B is finite then $A \cap B$ is finite because $A \cap B \subseteq A$ and $A \cap B \subseteq B$. More generally, if C is a set of sets at least one member of which is finite then $\bigcap C$ is finite, because it's a subset of that member and subsets of finite sets are finite. Similarly, if A is finite then $A \setminus B$ is finite for any set B because $A \setminus B$ is a subset of A.

Before proving that the union of two finite sets is finite it will be useful to introduce another notion, that of a small set. Small will turn out to mean the same as finite, but it has a slightly different definition. We say that A is small if for every non-empty set B there is a $D \in B$ such that if $C \in B$ and $C \subseteq D$ then $A \cap C = A \cap D$. Every finite set is small. We can see this as follows. If B is a non-empty set of subsets of A then there is, by the smallness of A a $D \in B$ such that if $C \in B$ and $C \subseteq D$ then $A \cap C = A \cap D$. Both C and D are members of B and hence are subsets of A so $A \cap C = C$ and $A \cap D = D$ so we can equally well say that there is a $D \in B$ such that if $C \in B$ and $C \subseteq D$ then $C = D$. In other words B has a minimal member. We've already seen that that if every non-empty set of subsets of A has a minimal member then A is finite. Conversely, suppose that A is finite and B is a non-empty set, not necessarily a set of subsets of A. Let E be the set of sets of intersections of members of B with A. In other words,

$$E = \{F \in [PA] : [\exists C \in B : [F = [C \cap A]]]\}.$$

E is a set of subsets of A. B is non-empty so there is a $C \in B$. Then $C \cap A \in E$, so E is non-empty. A is finite so every non-empty set of subsets of A has a minimal member. Let G be the minimal member of E. $G \in E$ so there is a $D \in B$ such that $G = D \cap A$. Suppose $C \in B$ and $C \subseteq D$. Let $F = C \cap A$. From $C \subseteq D$ it follows that $C \cap A \subseteq D \cap A$, i.e. that $F \subseteq G$. G is a minimal member of E and $F \subseteq G$ so $C = D$. In other words, $C \cap A = D \cap A$. We've just seen that for every non-empty set B there is a $D \in B$ such that if $C \in B$ and $C \subseteq D$ then $A \cap C = A \cap D$. In other words, A is small. We've just shown that finiteness implies smallness and we've

already seen that smallness implies finiteness so smallness and finiteness are equivalent.

We can now prove that the union of two finite sets is finite. Actually I'll prove that the union of two small sets is small, with smallness as defined in the preceding paragraph, but that then implies that the union of two finite sets is finite. Suppose then that A and B are small, and that C is a non-empty set. Since A is small there is an $E \in C$ such that $D \in C$ and $D \subseteq E$ imply $D \cap A = E \cap A$. Choose such an E. Let F be the set of members of C whose intersection with A is $E \cap A$. In other words,

$$F = \{D \in C : D \cap A = E \cap A\}.$$

Now $E \in F$ so F is non-empty. B is small so there is an $H \in F$ such that $G \in F$ and $G \subseteq H$ imply $G \cap B = H \cap B$. Choose such an H. Then $H \in F$ and so $H \in C$. Suppose $G \in C$ and $G \subseteq H$. Then $G \cap A \subseteq H \cap A$. But $H \in F$ so $H \cap A = E \cap A$ and therefore $G \cap A \subseteq E \cap A$. Since $G \in C$ it follows from the way that E was chosen that $G \cap A = E \cap A$. Therefore $G \in F$. From this and the way H was chosen it follows that $G \cap B = H \cap B$. Now

$$G \cap (A \cup B) = (G \cap A) \cup (G \cap B)$$

and

$$H \cap (A \cup B) = (H \cap A) \cup (H \cap B)$$

so from $H \cap A = E \cap A$ and $G \cap B = H \cap B$ we find that

$$G \cap (A \cup B) = H \cap (A \cup B).$$

What we've shown is that for any non-empty set C there is an $H \in C$ so that $G \in C$ and $G \subseteq H$ imply $G \cap (A \cup B) = H \cap (A \cup B)$. In other words $A \cup B$ is small. So the union of two small sets is small and therefore the union of two finite sets is finite.

From this and the fact that $\{x\}$ is finite we find that if A is finite then so is $A \cup \{x\}$ for any x. One consequence of this that if every proper subset of a set is finite then the set itself is finite. Indeed, suppose B is a set such that every proper subset of B is finite. Either B is empty or there is some $x \in B$. If B is empty then we're done, because we already know the empty set is finite. If $x \in B$ then let $A = B \setminus \{x\}$. Then $A \subseteq B$ and x is a member of B but not of A so A is a proper subset of B and therefore is finite. But we just saw that if A is finite then so is $A \cup \{x\}$, which in this case is B, so B is finite.

Induction for finite sets

The following is the counterpart for finite sets to the principle of mathematical induction for integers:

Suppose A is a finite set and B is a set of sets such that $\emptyset \in B$ and for all $C \in B$ and $x \in A$ we have $C \cup \{x\} \in B$. Then $A \in B$.

To prove this, first set $D = B \cap PA$. Then D is a set of subsets of A. It's non-empty because $\emptyset \in D$. It therefore has a maximal member. Let E be such a member. E is a subset of A. If it were a proper subset there would be an x which is in A but not in E. Let $F = E \cup \{x\}$. Since $E \subseteq A$ and $\{x\} \subseteq A$ we have $F \subseteq A$. Also, $E \in D$ so $E \in B$. From the properties which B was assumed to have it follows that $E \cup \{x\} \in B$, i.e. that $F \in B$. Now $D = B \cap PA$ and $F \in B$ and $F \in PA$ so $F \in D$. E is a proper subset of F since x is a member of F but not of E. But E was a maximal member of D. This is impossible, so our assumption that E is a proper subset of A is untenable.

The statement above has a sort of converse:

Suppose A is a member of every set of sets B such that $\emptyset \in B$ and for all $C \in B$ and $x \in A$ we have $C \cup \{x\} \in B$. Then A is finite.

To prove this we just take B to be the set of finite subsets of A. We've already proved that it has the required properties. By what we've just proved it follows that $A \in B$ and therefore that A is finite.

We can use induction on sets to generalise our earlier theorem about the union of two finite sets being finite to finite unions of finite sets. Suppose A is a finite set and each member of A is also a finite set. Let B be the set of subsets of A such that $\bigcup B$ is finite. We have $\bigcup \emptyset = \emptyset$ and $\emptyset$ is finite so $\emptyset \in B$. If $C \in B$ and $D \in A$ then

$$\bigcup [C \cup \{D\}] = [\bigcup C] \cup D$$

and $\bigcup C$ and D are both finite so $\bigcup [C \cup \{D\}]$ is finite. In other words, if $C \in B$ and $D \in A$ then $\bigcup [C \cup \{D\}] \in B$. The set B therefore satisfies the conditions from our induction principle for sets and we can therefore conclude that $A \in B$, i.e. that $\bigcup A$ is finite.

Another thing we can prove by induction is that the power set of a finite set is finite. For this we first need a preliminary lemma saying that if PA is

finite then for any x the set $B = P[A \bigcup \{x\}] \setminus PA$ is also finite. Either x is a member of A or it isn't. If it is then $B = \emptyset$ and we've already seen that $\emptyset$ is finite. Suppose then that x is not a member of A. If C is a non-empty set of subsets of B then we construct a set D of sets of subsets of PA by saying that $E \in D$ if and only if $E \cup \{x\} \in C$. C was assumed to be non-empty so there is an F in C. Then $F \setminus \{x\} \in D$ so D is also non-empty. A is finite so D has a minimal member. Let G be such a member. Then $G \cup \{x\}$ is a minimal member of C. So every non-empty set C of subsets of B has a minimal member and therefore B is finite.

We've just seen that if PA is finite then so is $P[A \cup \{x\}] \setminus PA$ for any x. But

$$P[A \cup \{x\}] = PA \cup [P[A \cup \{x\}] \setminus PA]$$

and the union of two finite sets is finite so $P[A \cup \{x\}]$ is finite. Now $P\emptyset = \{\emptyset\}$ is finite so by induction on sets we can conclude that if B is finite then so is PB.

For reference here are the main finiteness properties we've proved so far:

- $\emptyset$ is finite, as is $\{x\}$ for any x.
- If $A \subseteq B$ and B is finite then so is A.
- If A is finite then so is $A \setminus B$ for any B.
- If A is a set of sets at least one of which is finite then $\bigcap A$ is finite. In particular $B \cap C$ is finite if B or C is finite.
- If A is a finite set of sets all of which are finite then $\bigcup A$ is finite. In particular $B \cup C$ is finite if B or C is finite.
- If A is finite then so is PA.

Induction can also be used to show that the Cartesian product of finite sets is finite, but first we'll need to define Cartesian products.

Lists

The main goals of this section is make precise what we mean by a list, and to define various other useful objects in terms of them.

Kuratowski pairs

We start with ordered pairs. There is a simple way to define ordered pairs which unfortunately does not generalise to ordered triples or beyond, but which we can use as a starting point to construct lists.

What do we want from such a construction?

- Given any x and y, not necessarily distinct, we should be able to define the pair (x, y).
- Given any z we should be able to determine whether it is an ordered pair, i.e. whether there exist x and y such that $z = (x, y)$.
- Given any z which is an ordered pair we should be able to find x and y such that $z = (x, y)$.
- The x and y above should be uniquely determined by z. In other words if also $z = (v, w)$ for some v and w then it must be the case that $v = x$ and $w = y$.

With these considerations in mind, we define the Kuratowski pair of x and y as

$$«x, y» = \{\{x\}, \{x, y\}\}.$$

I've avoided writing this as (x, y) for two reasons. First, we don't yet know that it has the properties listed above. Second, even if it does there might be other constructions which have the same properties and we shouldn't prematurely commit ourselves to a particular implementation of ordered pairs.

The first property above is fairly obvious. For every x and y the set $«x, y»$ does indeed exist, as a simple consequence of the Axiom of Pairing. The second is much less obvious. What properties does $«x, y»$ have which set it apart from things which are not Kuratowski pairs?

- $«x, y»$ is a non-empty set of non-empty sets.
- $\bigcap «x, y»$ has exactly one member.
- $[\bigcup «x, y»] \setminus [\bigcap «x, y»]$ has at most one member.

These are all properties which we can express in our language. We can say that a set A has at least one member, even though we don't have a notation

for numbers, by saying $\exists w.w \in A$. Similarly, we can say that A has at least one member by saying that any two members must be equal:

$$\forall u \in A.\forall v \in A.[u = v].$$

We can say that a set has exactly one member by taking these two statements and joining them with an $\wedge$.

Each statement is true. The first is clear. The second and third follow from $\bigcap «x, y» = \{x\}$ and $\bigcup «x, y» = \{x, y\}$, so $[\bigcup «x, y»] \setminus [\bigcap «x, y»]$ is either $\emptyset$ or $\{y\}$, according to whether $x = y$ or not.

Is it true that if z satisfies the three conditions

- z is a non-empty set of non-empty sets,
- $\bigcap z$ has exactly one member, and
- $[\bigcup z] \setminus [\bigcap z]$ has at most one member

then z is an ordered pair? Yes. Suppose z is such a set. Let x be the only member of $\bigcap z$. Let y be the only member of $[\bigcup z] \setminus [\bigcap z]$, if there is one, and let it be the only member of $\bigcap z$ otherwise. We always have $\bigcap z \subseteq \bigcup z$ so

$$\bigcup z = [\bigcap z] \cup [[\bigcup z] \setminus [\bigcap z]].$$

Now $\bigcap z = \{x\}$ and either $\bigcap z = \{y\}$ or $\bigcap z = \emptyset$ and $[\bigcup z] \setminus [\bigcap z] = \{y\}$ or $\bigcap z = \{y\}$. In either case we have

$$\bigcap z = \{x\}$$

and

$$\bigcup z = \{x, y\}.$$

Any member of z is a subset of $\bigcup z$ and a superset of $\bigcap z$ but the only sets like that are $\{x\}$ and $\{x, y\}$, so

$$z \subseteq \{\{x\}, \{x, y\}\}$$

or

$$z \subseteq «x, y».$$

It's easy to check that any proper subset of $«x, y»$ would violate at least one of the conditions above so

$$z = «x, y»$$

This shows that the second of our three properties holds, i.e. that we can identify which sets are Kuratowski pairs. It also shows that the third property holds though, since we've uniquely identified x and y from the pair. I'll refer to x as the left component and y as the right component.

Kuratowski triples?

You might reasonably hope to generalise the construction above to triples by defining

$$«x, y, z» = \{\{x\}, \{x, y\}, \{x, y, z\}\}.$$

This turns out not to work though. With this definition

$$«v, v, w» = \{\{v\}, \{v, v\}, \{v, v, w\}\}$$

so

$$«v, v, w» = \{\{v\}, \{v, w\}\}$$

and

$$«v, w, w» = \{\{v\}, \{v, w\}, \{v, w, w\}\}$$

so

$$«v, w, w» = \{\{v\}, \{v, w\}\}$$

and hence

$$«v, v, w» = «v, w, w».$$

In other words, we have no way of identifying the middle component of the triple. This is in contrast to the situation with pairs, where we could identify the left and right components from the pair.

Lists

We'd like to have not just pairs and triples but lists of arbitrary finite length. The Kuratowski construction works for pairs, but not for length greater than two. We can still use it to define lists though. The idea is to define lists as sets of Kuratowski pairs. Each member of the set will have as its left component an element of the list and as it's right component the list of all subsequent elements. An empty list is just the empty set. A list with one element, say z will be a set whose only member is a pair with z as its left

component and the list of all subsequent elements, which is just the empty list, as it's right component. In other words,

$$() = \emptyset$$

and

$$(z) = \{\langle\langle z, ()\rangle\rangle\}.$$

Similarly, a list with two elements, say y and z is a set with two members, one of which is the pair from (z) and the other of which is a pair with y as its left component and the set (z) as its right component,

$$(y, z) = \{\langle\langle y, (z)\rangle\rangle, \langle\langle z, ()\rangle\rangle\}.$$

Similarly for lists with three elements.

$$(x, y, z) = \{\langle\langle x, (y, z)\rangle\rangle, \langle\langle y, (z)\rangle\rangle, \langle\langle z, ()\rangle\rangle\}.$$

You can expand these out, but it rapidly becomes messy.

$$(x, y, z) = \{\langle\langle x, \{\langle\langle y, \{\langle\langle z, \emptyset\rangle\rangle\}\rangle\rangle, \langle\langle z, \emptyset\rangle\rangle\}\rangle\rangle, \langle\langle y, \{\langle\langle z, \emptyset\rangle\rangle\}\rangle\rangle, \langle\langle z, \emptyset\rangle\rangle\}.$$

To write this fully in our language of course we should proceed further and write each of the seven Kuratowski pairs above as a set of sets, as in the definition of Kuratowski pairs. Fortunately we don't really need to do that. Even the expansion above isn't ever needed. We can think of lists as being built from the empty list and prepending elements one at a time. The empty list is the empty set. To prepend an element x to a list A we form the set

$$A \cup \{\langle\langle x, A\rangle\rangle\},$$

which is

$$A \cup \{\{\{x\}, \{x, A\}\}\}.$$

Everything you might want to do with lists can be done using three basic operations. One is the prepending operation described above, which adds a pair to an existing list. The other two extract the left and right components of the most recently added pair. How do we know which pair was added most recently? It's the one whose right component is a superset of the right components of the others. Taking the left component of this pair gives the first element of the list. Taking its right component gives the result of

removing that element from the list. These only work on non-empty lists, of course.

As an example of decomposing a list operation into these three basic operations, consider reversing a list. We start with the list we want to reverse and an empty list. One by one we remove elements from the list we started with and prepend them to the list which was initially empty until the list we started with has been emptied.

As another example, consider the problem of concatenating two lists, which I'll call the left list and right list, to distinguish the positions of their elements in the combined list. We start by reversing the left list. We then removed elements from it one at a time and prepend them to the right list. We continue until the left list is empty. The right list will then be their concatenation.

I've chosen a representation where new elements are added to a list from the left. I could equally have chosen one where we add elements from the right, appending rather than prepending. This particular choice of representation for lists is borrowed from LISP though and for historical reasons the choice there is that the newest list element is the first rather than the last. The three basic list operations described above have the names `cons`, `car` and `cdr` in LISP, again for historical reasons. Racket is a descendent of LISP and those functions all put in a brief appearance in the Racket program in the introductory chapter.

Interfaces and implementation

An important principle in computer science is that of abstraction through interfaces. The idea is to separate the interface from the details of its implementation. A user should be able to rely on the public interface of a system without needing to know any of the details of how it is implemented. In fact, using details of the implementation which are not part of the public interface is strongly discouraged.

In our case we have a notion of lists and basic list operations, like adding an element to the list, extracting the most recently added element and extracting the remainder of the list. The particular implementation may be rather complicated but should never be needed and indeed should never be used beyond proving a small number of basic properties, the fact that

if we add an element and then remove it then we are left with the same list we started with, and that the element we extracted is the one we added. Any more complicated operations, like reversal or concatenation should be defined in terms of the basic operations and any properties of the those more complicated operations should be proved from the properties of the basic operations, without reference to the implementation. This is done partly to reduce cognitive load and partly to allow the implementation to be replaced by a different one without any need for external changes.

There's a difference between mathematics and computer science, but it's one which makes the principle of separating the interface and implementation even more useful in mathematics than in computer science. The two subjects have different notions of an efficient implementation. The efficiency of an implementation of lists in a programming language or library is mostly a matter of resource usage, and the most important resource is usually time. A program which uses lists won't need to be rewritten if the list implementation changes, provided it relies only on the public interface and not on the details of the previous implementation, but it may well run faster, so there is an obvious benefit to replacing a simple but slow implementation with a more complicated but faster one. For an implementation of lists in set theory the situation is different. What makes an implementation efficient is the ease with which we can prove that the basic operations have the required properties. Once this is done there's no real advantage to replacing it with a different implementation. The same applies to abstraction through interfaces in general. Separating interface and implementation is useful in both subjects, but in mathematics efficiency of implementation is a pedagogical issue rather than a practical one.

One advantage of the abstract approach described above is that often we realise that two different interfaces are in fact the same, except for names. Stacks appeared as a data structure in the introductory chapter. I didn't define them precisely there but the basic operations on a stack are pushing something on to it and popping something off. An implementation of a stack just needs to provide these operations and ensure that the item popped off is the same as the one which was pushed and that the stack after a push and a pop is the same as it was before. If this sounds similar to the rules for lists then that's because they are, except for terminology, identical. A stack is just a list. In a computing application you might or might not want to implement a stack as a list but from a mathematical point of

view there's no reason not to.

Pairs again

There are a number of ways we could define ordered pairs in set theory. The usual choice is Kuratowski pairs. An equal valid choice would be Kuratowski pairs with the left and right components swapped. The choice of which to call left and which to call right was entirely arbitrary. There are a number of other options available. The particular choice I'm going to make here is to regard ordered pairs as lists with two elements. As long as we only use the basic operations and properties it doesn't matter which implementation we choose. One minor advantage of using lists of length two though is that we can now entirely forget the notation «x, y».

Cartesian products

The set of ordered pairs (x, y) with $x \in A$ and $y \in B$ is called the Cartesian product of A and B, written $A \times B$. In the common special case where $A = B$ we often write A^2 rather than $A \times A$. In a similar way we define A^3 to be the list of ordered triples, i.e. lists of the form (x, y, z), where each of x, y and z is an element of A.

$A \times B$ is indeed a set, as are A^2 and A^3. This is less obvious than it might seem, but still true.

If A and B are finite sets then $A \times B$ is finite. This is proved by a double induction on sets. Suppose $x \in A$. Let D be the set of subsets C of B such that $\{x\} \times C$ is finite. $\{x\} \times \emptyset$ is $\emptyset$, which is finite, so $\emptyset \in D$. If $C \in D$ and $y \in B$ then

$$\{x\} \times [C \cup \{y\}] = [\{x\} \times C] \cup \{(x, y)\}.$$

This is the union of two finite sets and so is finite. In other words, $\emptyset \in D$ and if $C \in D$ and $y \in B$ then $C \cup \{y\} \in D$. B is finite so by induction $B \in D$. In other words $\{x\} \times B\}$ is finite. Now let F be the set of all subsets E of A such that $E \times B$ is finite. $\emptyset \times B$ is $\emptyset$, which is finite, so $\emptyset \in F$. If $x \in A$ then, as we just saw, $\{x\} \times B\}$ is finite. Suppose $E \in F$.

$$[E \cup \{x\}] \times B = [E \times B] \cup [\{x\} \times B]$$

so $[E \cup \{x\}] \times B$ is the union of two finite sets and so is finite. In other words, $\emptyset \in F$ and if $E \in F$ and $x \in A$ then $E \cup \{x\} \in F$. A is finite so by induction $A \in F$. In other words, $A \times B$ is finite.

In particular, if A is finite then so are A^2 and A^3.

Relations

A binary relation is a set of ordered pairs. From now on I'll just use relation as shorthand for binary relation unless otherwise specified since we're mostly concerned with binary relations. The definition above is too general to be of much use. We really need to impose more conditions to get any interesting properties but there are a few useful definitions that make sense in this level of generality.

Basic definitions

A relation R is called diagonal if $(x, y) \in R$ implies $x = y$. For any set A we can define the relation I_A as the set of all ordered pairs (x, x) for $x \in A$. This is a diagonal relation and is called the diagonal relation on A. These are in fact the only examples of diagonal relations.

The domain of a relation R is the set of x such that there is a y with $(x, y) \in R$. The range of R is the set of y such that there is an x with $(x, y) \in R$. The range and domain of I_A are just A.

The inverse of a relation R is the set of all ordered pairs (x, y) such that $(y, x) \in R$. I'll denote it by R^{-1}. Note that R^{-1} is a set of ordered pairs so it is also a relation. We can therefore take its inverse, ${R^{-1}}^{-1}$. Now $(x, y) \in {R^{-1}}^{-1}$ if and only if $(y, x) \in R^{-1}$, which happens if and only if $(x, y) \in R$. By the Axiom of Extensionality it follows that

$$ {R^{-1}}^{-1} = R. $$

Given two relations R and S we can define their composition, which is written $R \circ S$, defined to be the set of ordered pairs (x, z) such that there is a y with $(x, y) \in S$ and $(y, z) \in R$. The name is standard and the notation somewhat standard, but most authors reverse the roles of R and S in the

definition. The problem with doing that is that functions, as we'll see, are a kind of relation and the standard notation for composition of functions writes them in reverse order, i.e. $f \circ g$ is the result of applying g and then f. To accommodate this convention, which is unfortunate but too well established to attempt to change, it's necessary to do composition of relations in the reverse order as well. The composition of relations is also a relation, and so can be composed with other relations. This has the associativity property

$$(R \circ S) \circ T = R \circ (S \circ T).$$

If the domain of R is a subset of A then $R \circ I_A = R$. If the range of R is a subset of A then $I_A \circ R = R$.

Another useful identity is

$$(R \circ S)^{-1} = (S^{-1}) \circ (R^{-1}).$$

A relation R is said to be symmetric if $R = R^{-1}$, i.e. if $(x, y) \in R$ if and only $(y, x) \in R$. It's said to be transitive if $R \circ R \subseteq R$, i.e. if $(x, z) \in R$ whenever $(x, y) \in R$ and $(y, z) \in R$. The diagonal relation on a set is always symmetric and transitive. If R is transitive then so is R^{-1}.

A relation R is said to be antisymmetric if $R \cap R^{-1}$ is diagonal or, equivalently if $(x, y) \in R$ and $(y, x) \in R$ imply $x = y$. The terminology is unfortunate since antisymmetric is not the opposite of symmetric. A relation can be symmetric and antisymmetric. Diagonal relations, for example, are both symmetric and antisymmetric. It's also possible for a relation to be neither symmetric nor antisymmetric. Note that if R is antisymmetric then so is R^{-1}.

A relation R is said to be left unique if $R^{-1} \circ R$ is diagonal. In other words, if $x = z$ whenever there is a y such that $(x, y) \in R$ and $(y, z) \in R^{-1}$ or, equivalently, whenever $(x, y) \in R$ and $(z, y) \in R$. In other words, for any y there is at most one ordered pair has y as its right element. Similarly R is said to be right unique if $R \circ R^{-1}$ is diagonal, which is equivalent to saying that for any x there is at most one ordered pair with x as its left element. This may seem backwards but this use of left and right is standard.

If the relation R is a subset of the Cartesian product $A \times B$ then we say that it's a relation from A to B and if R is a subset of $A \times A$ then we say that it's

a relation on A. If R is a relation from A to B then R^{-1} is a relation from B to A. In particular if R is a relation on A then so is R^{-1}. If R is a relation from B to C and S is a relation from A to B then $R \circ S$ is a relation from A to C. In particular if R and S are relations on A then so is $R \circ S$.

Examples

As examples of the properties above, consider the following relations on the set of natural numbers:

- R is the set of (x, y) with $x = y$.
- S is the set of (x, y) with $x \leq y$.
- T is the set of (x, y) with $x < y$.
- U is the set of all (x, y).
- V is the set of (x, y) with $x \neq y$.

The domain of R, S, T, U, and V is the set of natural numbers. The range is also the set of natural numbers in each case, except that of T, whose range is the set of positive integers since every positive integer is greater than some natural number and every natural number which is greater than some natural number is a positive integer.

R is diagonal. None of the other relations are. It is also the only one which is left or right unique.

Now

- R^{-1} is the set of (x, y) with $x = y$, i.e. just R, so R is symmetric and antisymmetric. As mentioned above, diagonal relations are always symmetric and antisymmetric.
- S^{-1} is the set of (x, y) with $x \geq y$, which is not the same as S, so S is not symmetric. $S \cap S^{-1} = R$ and R is diagonal so S is antisymmetric.
- T^{-1} is the set of (x, y) with $x > y$, which is not the same as T, so T is also not symmetric. $T \cap T^{-1} = \emptyset$ and $\emptyset$ is diagonal so T is antisymmetric.
- U^{-1} is the set of all (x, y), which is the same as U, so U is symmetric. $U \cap U^{-1} = U$ and U is not diagonal, so U is not antisymmetric.

- V^{-1} is the set of (x, y) with $x \neq y$, which is the same as V, so V is also symmetric. $V \cap V^{-1} = V$ and V is not diagonal so V is not antisymmetric.

and

- $R \circ R$ is the set of (x, y) with $x = y$, i.e. R, which is a subset of R, so R is transitive. As mentioned above diagonal relations are always transitive.
- $S \circ S$ is the set of (x, y) with $x \leq y$, i.e. S, so S is transitive.
- $T \circ T$ is the set of (x, y) with $x + 1 < y$, which is a subset of T, so T is transitive. Note that $T \circ T$ is a proper subset of T, unlike what we saw for R and S, but the relation is still transitive.
- $U \circ U$ is the set of all (x, y), i.e. U, so U is transitive.
- $V \circ V$ is the set of all (x, y), i.e. U, which is not a subset of V so V is not transitive.

Most of these are fairly straightforward. If $(x, z) \in S \circ S$ then there is a y such that $(x, y) \in S$ and $(y, z) \in S$, i.e. such that $x \leq y$ and $y \leq z$. It follows that $x \leq z$, i.e. that $(x, z) \in S$. So $S \circ S \subseteq S$. This is all we need for transitivity, but if we want to prove the statement made above that $S \circ S = S$ then we also need to show the reverse inclusion $S \subseteq S \circ S$. In other words we need to show that if $x \leq z$ then there is a y such that $x \leq y$ and $y \leq z$. This is easy. Either $y = x$ or $y = z$ will work. The argument for T is similar. If $(x, z) \in T \circ T$ then there is a y such that $(x, y) \in T$ and $(y, z) \in T$, i.e. such that $x < y$ and $y < z$. It follows that $x < z$, i.e. that $(x, z) \in T$. So $T \circ T \subseteq T$ and T is transitive. To prove the stronger statement given above we note that since we're dealing with natural numbers $x < y$ and $y < z$ imply $x + 1 \leq y$ and $y + 1 \leq z$, from which we get $x + 2 \leq z$ and then $x + 1 < z$. To see that $V \circ V = U$, note that if $(x, z) \in U$ then there is a natural number y distinct from x and z. To be more concrete, the numbers 0, 1, and 2 are all distinct so at least one of them is unequal to either x or z. Call the least such number y. Then $(x, y) \in V$ and $(y, z) \in V$ so $(x, z) \in V \circ V$. So $U \subseteq V \circ V$. The reverse inclusion is trivial since every relation on the natural numbers is a subset of U, essentially by definition.

Functions

If R is a relation from A to B then the domain of R is a subset of A and the range of R is a subset of B. We say that R is left total if the domain of R is all of A and that it's right total if the range of R is all of B. It follows that R^{-1} is left total if R is right total and vice versa.

Some properties of a relation from A to B depend only on the relation, i.e. the set of ordered pairs and others depend on the sets A and B. Left and right uniqueness, for example, depend only on the relation while left and right totality depend on A and B as well.

A relation which is left total and right unique is called a function. If F a function from B to C and G is a function from A to B then $F \circ G$ is a function from A to C. It's possible for $F \circ G$ to be a function without F or G begin functions though. The precise conditions needed are

- For each x in A there is a y in the domain of G such that $(x, y) \in F$.
- If $(x, y) \in F$, $(y, z) \in F$, $(x, w) \in F$ and $(w, v) \in F$ then $v = z$.

Although this might seem like an obscure way to construct a function compared to composition of functions it is in fact frequently used.

Every function is left total by definition. Functions which are also right total are called surjective. Every function is right unique by definition. Functions which are also left unique are called injective. Functions which are both right total and left unique are called bijective, or invertible. Every function is a relation and so has an inverse, which is also a relation. For bijective relations this inverse relation is also a function. If F is a bijective function from A to B then $F \circ F^{-1} = I_B$ and $F^{-1} \circ F = I_A$.

If you're accustomed to thinking of functions as being defined by algorithms then the definition above does not correspond to your intuition. Different algorithms can certainly give the same function. For example, taking a number and adding it to itself and taking a number and multiplying it by two are different algorithms but they correspond to the same function according to the definition above. Later we will see that there are functions for which there is no corresponding algorithm as well. If you're used to thinking of functions in terms of graphs, on the other hand, then the definition above is exactly your intuition. Functions are simply defined as

graphs. Note that this is the one place in these notes where I use the word graph in the sense that it's used in algebra and calculus. Everywhere else it will by used in the same sense as in graph theory.

There are relatively few terminological conflicts between computer science and related fields like mathematics, logic and linguistics. Sometimes computer scientists use a different term for the same concept but its rare for them to use the same term for a different concept. This is unfortunately one of the exceptions. Computer scientists refer to the algorithmic notion of functions as functions. What word do they use for the graph notion of functions? Also function! This is confusing, but less of a problem than it might appear since the algorithmic notion is much more common. Logicians are the only people who have an adequate terminology. They refer to the algorithmic notion as intensional functions and the graph notion as extensional functions. Function without an adjective normally means extensional unless otherwise specified. Note that intensional is not a typo for intentional. The term from logic has an s rather than a t.

A useful fact about finite sets is that if A is a finite set and F is an injective function from A to A then F is also a surjective function from A to A. This can be proved by set induction. The only function from $\emptyset$ to $\emptyset$ is $\emptyset$, because there are no ordered pairs (x, y) with $x \in \emptyset$ and $y \in \emptyset$. Now $\emptyset$ is trivially right total so every injective function $\emptyset$ to $\emptyset$ is a surjective function from $\emptyset$ to $\emptyset$. It therefore suffices to prove that if every injective function from A to itself is surjective then every injective function from $A \cup \{x\}$ to itself is surjective. This is certainly true if $x \in A$ so we can limit our attention the case where x is not a member of A. Assume then that F is an injective function from $A \cup \{x\}$ to itself and x is not a member of A. F is left total so there is a $y \in A$ such that $(x, y) \in F$. F is left unique so there is no $w \in A$ such that $(w, y) \in F$. F is left total and right unique so for all $w \in A$ there is a unique $z \in A \cup \{x\}$ such that $(w, z) \in F$. If $y = x$ then this z is not x and so must be in A. In this case the set of pairs (w, z) with $w \in A$ is an injective function from A to itself. It must therefore be surjective. There is then, for each $z \in F$ a $w \in A$ such that $(w, z) \in F$. F is injective so we can't have $(x, z) \in F$. and therefore $y = x$. In other words, $x = y$ if and only if there is, for each $w \in A$, a $z \in A$ such that $(w, z) \in F$, and in this case every member of A is in the range of F and so is x so F is a surjective function from $A \cup \{x\}$ to itself. It remains to consider the case where y is not equal to x, and so is member of A, and there is some $w \in A$ for which there is no $z \in A$ with

$(w, z) \in F$. F is left total so there is some $z \in A \cup \{x\}$ with $(w, z) \in F$ and so we must have $z = x$ for such w. Since F is left unique there is at most one such w and we already know there's at least one so there must be exactly one. Let

$$G = F \cup \{(w, y)\} \setminus \{(w, x), (x, y)\}.$$

This is an injective function from A to itself and so is also a surjective function. So for all $z \in A$ there is a $v \in A$ such that $(v, z) \in G$. If z is not y then $(v, z) \in F$ so z is in the range of F. If z is y then z is also in the range of F because then $(x, z) \in F$. So all members of A are in the range of F. x is also in the range of F since $(w, x) \in F$ so all of $A \cup \{x\}$ is in the range of F, which therefore must be surjective. F was an arbitrary injective function from $A \cup \{x\}$ to itself so all injective functions from $A \cup \{x\}$ to itself are surjective. We've shown that all injective functions from $\emptyset$ to itself are surjective and that if all injective functions from A to itself are surjective then all injective functions from $A \cup \{x\}$ to itself are surjective. By induction on sets it follows that all injective functions from a finite set to itself are surjective.

Replacement

As was mentioned earlier, it would be nice to be able to define a set by giving a Boolean expression with a single free variable and define a set as those values of the variable for which the expression is true, but this doesn't work because we could use it to show the existence of the set of all sets, which leads to contradiction. That's why the Axiom of Separation has the form that it does.

Similarly, we would like to be able to define a function by giving a Boolean expression with two free variables, one representing the argument of the function and one representing the value of the function at that argument. We would need some additional restrictions to make sure that the relation we get in this way is left total and right unique, but we also need to use some set as the domain, to avoid accidentally creating the set of all sets. We could, for example, assume the following.

- Replacement Suppose P is an expression in which x and y are free variables and A and F do not appear. For all A the statement that for each $x \in A$ there is unique y such that P holds implies that there is a set F such that $(x, y) \in F$ if and only if $x \in A$ and P holds.

Formally, the statement that for each x there is unique y such that P holds is the result of linking with a $\wedge$ the two statements

$$[\forall x \in A.[\exists y.P]]$$

and

$$[\forall x \in A.[\forall y.[\forall z.[[P \wedge Q] \supset [y = z]]]]]$$

where Q is P with all free occurrences of y replaced by z. We can simplify this a little by combining the $\forall x \in A$'s in the two statements:

$$[\forall x \in A.[[\exists y.P] \wedge [\forall y.[\forall z.[[P \wedge Q] \supset [y = z]]]]]].$$

The first statement is what gives us left totality and the second is what gives us right uniqueness. For this to work we need to make the additional assumption that z does not appear in P. We can simplify this a little by combining the $\forall x \in A$'s in the two statements:

$$[\forall x \in A.[[\exists y.P] \wedge [\forall y.[\forall z.[[P \wedge Q] \supset [y = z]]]]]]$$

The statement about F is the result of joining

$$[\forall x.[\forall y.[[[x \in A] \wedge P] \supset [(x,y) \in F]]]]$$

and

$$[\forall x.[\forall y.[[(x,y) \in F] \supset [[x \in A] \wedge P]]]]$$

Again, we can simplify this slightly by combining the quantifiers they have in common:

$$[\forall x.[\forall y.[[[[x \in A] \wedge P] \supset [(x,y) \in F]] \wedge [[(x,y) \in F] \supset [[x \in A] \wedge P]]]]].$$

The full formal version is then

$$[\forall A.[[\forall x \in A.[[\exists y.P] \wedge [\forall y.[\forall z.[[P \wedge Q] \supset [y = z]]]]]] \\ \supset [\exists F.[\forall x.[\forall y.[[[[x \in A] \wedge P] \supset [(x,y) \in F]] \wedge [[(x,y) \in F] \supset [[x \in A] \wedge P]]]]]]]].$$

Like Separation, Replacement is really an axiom schema rather than a single axiom. There is one axiom for each choice of P.

For finite sets we don't need a new axiom since we can prove the statement above from the axioms we already have by set induction. For infinite sets

though this doesn't follow from our existing axioms. In contrast to Separation, which is used all the time, Replacement is almost never used outside of set theory, and rarely used even within it. It is fairly plausible though and it can be shown that if the system is sound without it then it is also sound with it so we can be sure that we haven't accidentally introduced contradictions if we adopt it. This would not be the case if we didn't require the domain of the function to be specified.

The axiom schema above isn't quite the usual one. The usual choice gives the existence of the range of F rather than F itself. This is equivalent, in the sense that from either axiom we can derive the other. The usual choice has the advantage that we don't need to mention ordered pairs, just sets, and so the axiom schema can be introduced much earlier, before we've defined ordered pairs. On the other hand the axiom schema isn't needed earlier and the version with functions makes the motivation for its introduction much clearer.

Order relations, equivalence relations

A relation R on a set A is called reflexive if $I_A \subseteq R$, i.e. if $(x, x) \in R$ for all $x \in A$. Note that if R is reflexive then so is R^{-1}. Of our earlier examples R, S and U are reflexive while T and V are not.

A relation which is reflexive, transitive and antisymmetric is called a partial order. We just noted that if R is reflexive then so is R^{-1}. We've previously seen that if R is transitive then so is R^{-1} and that if R is antisymmetric then so is R^{-1}. It follows that if R is a partial order then so is R^{-1}. It's said to be a total order if in addition $R \cup R^{-1} = A \times A$, i.e. if for all $x \in A$ and $y \in A$ at least one of $(x, y) \in R$ or $(y, x) \in R$ holds.

Of our earlier example relations, R and S are partial orders and S is a total order. None of the others are partial orders and R is not a total order.

If we restrict a partial order to a subset then the result will always be a partial order on the subset and may or may not be a total order on the subset. If it is a total order then that subset is called a chain.

A relation R on a set A is said to be an equivalence relation if it is reflexive, transitive and symmetric. Of our earlier examples, both R and U are equivalence relations, while S, T and V are not. There is an important

equivalence relation, equivalence modulo n, on the set of natural numbers, defined for each natural number n. This is the set of ordered pairs (x, y) for which there is a natural number m such that either $x = y + m \cdot n$ or $x + m \cdot n = y$. The special case $n = 0$ gives the relation R from earlier and the special case $n = 1$ gives the relation U but the cases where $n > 1$, and particularly where n is prime, are more important.

Suppose R is a partial order on A. $y \in A$ is said to be a greatest member of A if $(x, y) \in R$ for all $x \in A$. $y \in A$ is said to be a maximal member if $z \in A$ and $(y, z) \in R$ imply $y = z$. $x \in A$ is said to be a least member of A if $(x, y) \in R$ for all $y \in A$. x is said to be a minimal member of A if $w \in A$ and $(w, x) \in R$ imply $w = x$.

If there is a greatest member then there is only one and it is also a maximal member. If there is a least member then there is only one and it is also a minimal member.

If A is a set of sets then $R = \{(B, C) \in A \times A : B \subseteq C\}$ is an order relation since $B \subseteq B$ for all $B \in A$, $B \subseteq D$ if $B \subseteq C$ and $C \subseteq D$, and $B \subseteq C$ and $C \subseteq B$ imply $B = C$.

As an example of the definitions above, suppose A is the set of non-empty finite subsets of some non-empty set E.

For any $x \in E$ we have $\{x\} \in A$. $\{x\}$ is in fact a minimal member since if B is a finite non-empty subset of $\{x\}$ then $B = \{x\}$. If x is the only member of E then $\{x\}$ is also a least member of A, but if there is some $y \in E$ with $x \neq y$ then A has no least member. A least member would have to be a subset of every member of A and hence a subset of both $\{x\}$ and $\{y\}$. The only set with this property is $\emptyset$, but it is not a member of A.

If E is finite then $E \in A$ and E is a greatest member of A since every member of A is a subset of E. If E is infinite then E is not a member of A and so can't be greatest member or maximal member. In fact there is no maximal member in this case and hence also no greatest member. Suppose B is a member of A. Then B is finite and E is infinite so B is not E. B is a member of A and all members of A are subsets of E so B is a subset of E and must be a proper subset since B is not equal to E. There is therefore some x which is a member of E but not of B. Let $C = B \cup \{x\}$. Then B is a subset of C and C is a subset of E. It's a finite subset. We proved that earlier. It's non-empty since $x \in C$. So $C \in A$. From this and $B \subseteq C$ it would follow that $B = C$ if

B were maximal, but x is a member of C and not of B so this is impossible. Therefore B is not maximal. Since B was an arbitrary member of A it follows that no member of A is maximal.

When defining finiteness earlier I used the terms minimal and maximal. You can check that the definitions given there agree with the definitions of minimal and maximal given above, with the relation being the set inclusion relation.

For any non-empty finite set A and partial order R on A there is a minimal member and a maximal member. This is proved by induction on sets. Let B be the set of subsets C of A such that C is empty or has a minimal and maximal member. Then $\emptyset \in B$. If $C \in B$ then $C \cup \{x\} \in B$ for all $x \in A$. This is proved as follows. If $C = \emptyset$ then x is both a minimal and maximal member of $C \cup \{x\}$. If C is not empty then it has a minimal and maximal member. Let z be a minimal member of C. Then $y \in C$ and $(y, z) \in R$ imply $y = z$. (x, z) either is or isn't a member of R. If it isn't then $y \in C \cup \{x\}$ and $(y, z) \in R$ imply $y = z$ so z is a minimal member of $C \cup \{x\}$. If $(x, z) \in R$ then for any $y \in C$ such that $(y, x) \in R$ we have $(y, z) \in R$ by the transitivity of R and so $y = z$, since z is a minimal member of C. But $(x, z) \in R$ so $(x, y) \in R$. R is antisymmetric so $(x, z) \in R$ and $(z, x) \in R$ imply $z = x$. In other words, whenever $y \in C$ such that $(y, x) \in R$ we have $y = x$. Therefore $y \in C \cup \{x\}$ and $(y, x) \in R$ imply $y = x$. In other words x is a minimal member of $C \cup \{x\}$. So either x or z is a minimal member of $C \cup \{x\}$. A similar argument shows that $C \cup \{x\}$ has a maximal member.

If R is an equivalence relation on a set A then we say that B is an equivalence class if B is a subset of A, for all $x \in B$ and $y \in B$ we have $(x, y) \in R$, and if $(x \in B)$ and $y \in B$ then $x \in B$.

Every element of A is a member of exactly one equivalence class. In fact, if $x \in A$ then $B = \{y \in A : (x, y) \in R\}$ is an equivalence class of which x is a member and if C is an equivalence class with $x \in C$ then $C = B$.

From a partial order we can construct an equivalence relation in a natural way. If R is a partial order on A then $S = R \cap R^{-1}$ is an equivalence relation. Another way to state this equation is to say that $(x, y) \in S$ if and only if $(x, y) \in R$ and $(y, x) \in R$.

Notation

You have no doubt noticed that this is not the usual way to write functions or relations. In place of $(x, y) \in F$ or $(x, y) \in R$ we usually write $y = F(x)$ or xRy. This is convenient, but dangerous. As I've mentioned before, first order logic does not cope well with meaningless expressions, like $F(x)$ where x is not in the domain of F. For this reason I'll be careful not to use the usual notation in this chapter, although I will use it in later chapters. You should be aware though that some rules of inference which are sound if we stick to the ordered pair notation become unsound when the usual functional notation is used. The most important of these is substitution. For real numbers, for example, we have the basic fact, known as the Law of Trichotomy, that

$$[\forall y.[[y < 0] \vee [y = 0] \vee [y > 0]]].$$

If we substitute the numerical expression $F(x)$ for y we get

$$[[F(x) < 0] \vee [F(x) = 0] \vee [F(x) > 0]]].$$

This is fine if x is in the domain of F but the usual way of interpreting a statement like $F(x) = 0$ is what we've written above as $(x, 0) \in F$, i.e. that x is in the domain of F and the value of F at x is 0. So the statement

$$[[F(x) < 0] \vee [F(x) = 0] \vee [F(x) > 0]]]$$

carries an implicit assumption that x is in the domain of F, which may not be true. This turns substitution from a mechanical process into one which requires actual thought, checking that the expressions which are given as arguments to functions represent values within the domains of those functions. Mathematicians generally consider the ease of use of the usual functional notation to be worth the extra work but it's important to realise that there is a trade-off here.

Natural numbers

Consider lists all of whose elements are $\emptyset$. One of these is the list of length 0, which is just $\emptyset$. If x is such a list then we can get another such list by prepending another $\emptyset$. In our representation of lists this is $x \cup \{(\emptyset, x)\}$ but in keeping with our policy of not relying on the particular representation

we can introduce the notation x' for the result of prepending a $\emptyset$ to x. Similarly, concatenating two such lists gives us another such list. Rather than writing down an expression for this in our particular representation of lists we can simply denote the concatenation of x and y by $x + y$. Rather than relying on the empty list being represented by the empty set we can write 0 for the empty list.

We the notation as above one can prove a number of elementary properties of these lists, such as the following.

- 0 exists.
- For all x, x' exists.
- For all x we have $x' \neq 0$.
- For all x we have $x + 0 = x$.

The second and fourth of these are proved by induction on sets. These may look familiar. They are in fact our first four axioms for elementary arithmetic. To get the remaining axioms we need to define more operations and relations. $x - y$, for example, can be defined to be the list, if there is one, which when concatenated with y gives x. $x \leq y$ will mean that there is a list which, when concatenated with x gives y and $x > y$ will mean that there isn't one. The only definition which is somewhat tricky is the definition of $x \cdot y$. To get $x \cdot y$ we start with a pair of lists where the first is initially 0 and the second is initially y. We then successively remove elements from the second list and concatenate copies of the x with the first. Once we've removed all elements of the second list the first list will be $x \cdot y$.

Once we have definitions for all the operations and relations it's not terribly difficult to show that the axioms of elementary arithmetic are all satisfied. We can also show that the three rules of inference from elementary arithmetic also apply to lists of $\emptyset$'s. Not surprisingly, the rule of induction for natural numbers follows from the induction theorem for finite sets. Note that induction for sets is a theorem though, rather than an axiom.

So we have a copy of elementary arithmetic sitting inside of set theory. There are other ways to embed arithmetic in set theory. The method above is not the most commonly used one. A more traditional approach is in terms of what are called the von Neumann ordinals. It's better not to think of the natural numbers as some particular representation. We shouldn't

therefore ask questions like whether $x \subset x''$. This happens to be true in the representation we've chosen, and also happens to be true in the representation in terms of von Neumann ordinals, but isn't true in some other representations of the natural numbers within set theory. In keeping with our policy of separating interface and implementation we should avoid applying operations or relations to natural numbers other than the ones from elementary arithmetic.

The particular construction chosen is somewhat arbitrary, but it's not randomly chosen either. Intuitively, lists have a length. Prepending and element increments the length. Concatenating lists adds their lengths. Of course many different lists will generally have the same length but if we want to choose a particular one for each length then we can do that by choosing some item and using only lists where all elements are equal to that item. The simplest choice for that item is $\emptyset$ because it's the only set whose existence is directly guaranteed by an axiom of set theory.

Infinite sets

I was very careful in the discussion above not to refer to the set of natural numbers, just to natural numbers. We have criteria, at least with the particular representation chosen, for determining whether something is a natural number, whether one natural number is the sum, difference or product of two other natural numbers, etc. Do we have a set of natural numbers though? We can write down a Boolean expression with a single free variable which evaluates to true if and only if a natural number, in our chosen representation, is substituted for all free occurrences of that variable. The Axiom of Separation, though, doesn't allow us to get sets from Boolean expressions; it only allows us to get subsets of a given set from them. Do we have a set which is large enough to contain at least the natural numbers, so that we then use Separation to find a set with the natural numbers and only the natural numbers?

With the axioms above we have no way to prove the existence of an infinite set. Extensionality doesn't give us any sets; it just says when two sets are equal. The axiom of elementary sets produces only finite sets, and in fact only sets with at most two elements. Separation gives us subsets of existing sets. We've already seen that subsets of finite sets are finite, so this axiom

can't give us an infinite set unless we already have one. We've also seen that power sets of finite sets are finite, so that axiom also can't give us an infinite set unless we already have one. The same is true of the Axiom of Union. Adding the Axiom of Replacement wouldn't help either since it's not difficult to show that functions with finite domains are finite.

We have a number of things though which, if they are sets, must be infinite. One is the natural numbers. If the natural numbers are a set then the set pairs (x, y) with $x \leq y$ is a partially ordered set with no greatest element. We've already seen that any partial order on a finite set has a greatest element, so the natural numbers can't be a finite set.

You might object that I've used the set of natural numbers in examples. Examples are meant to guide your intuition though so I sometimes presuppose things we don't yet know to be true. I've been careful to confine the natural numbers to examples though and not to use the existence of such a set in proving theorems.

Another set which, if is exits, must be infinite is that set of lists of items in a given non-empty set A. Any set which contains all such lists must be infinite, which we can see as follows.

Suppose A is a non-empty set and B is a set which contains all lists whose items are members of A. We can write down a Boolean expression which identifies which elements of B are actually lists all of whose items are members of A so we can use Separation to conclude that there is a set C whose members are precisely such lists. A is non-empty, so there is an $x \in A$. Let F be the set of pairs of lists (D, E) where E is the list obtained by appending x onto D. Then F is an injective function from C to itself. It is not surjective because the empty list $\emptyset$ is not in its range. But we've already shown that every injective function from a finite set to itself is surjective, so C cannot be finite. Subsets of finite sets are finite so B can't be finite either.

I introduced a notation earlier for the set of such lists. $[LA]$ denotes the lists all of whose elements are members of A. The existence of a notation for a set doesn't imply the existence of that set though, in much the same way that having a notation for the difference of two natural numbers doesn't imply that this difference exists for all pairs of natural numbers.

There are a number of ways to get infinite sets but all of them involve introducing some new axiom. We could just introduce an axiom saying that

there is an infinite set. This turns out not to be sufficient to establish the existence of all the infinite sets that we want. The usual procedure is to introduce an axiom establishing the existence of one particular infinite set which is in some sense large enough. We can, for example, use $L\{\emptyset\}$ for this purpose. As we saw, this is essentially the same as the set of natural numbers.

Once we have one infinite set it's fairly easy to get others, if we have the Axiom of Replacement. We can, for example, show the existence of LA for any set A. It's not terribly difficult to write down a Boolean expression which characterises LA in the sense that it has one free variable and evaluates to true if and only if the thing substituted for that variable is a list all of whose elements are members of A. By itself this isn't enough to show that LA exists since Separation only allows us to construct subsets of a given set from Boolean expressions. What we can do though is to construct another Boolean expression expression the fact that two lists are of equal length. Combining these two expressions we can write down a Boolean expression with two free variables which evaluates to true if and only if the first variable is a member of $L\{\emptyset\}$, i.e. a list all of whose elements are $\emptyset$'s and whose second variable is the set of all lists of members of A of length equal to the list in the first variable. Applying Replacement gives a function from $L\{\emptyset\}$ to the sets of lists of members of A of equal length. As the range of a function this is a set. Applying Union we finally get the set LA. This is actually the only time we will apply Replacement.

Motivated by the considerations above we'll take the following as an axiom:

- Infinity: The set $L\{\emptyset\}$ exists.

Cardinality

Set inclusion provides a notion of size of sets. A set is at least as large as any of its subsets and is strictly larger than any of its proper subsets. Inclusion is reflexive, transitive and antisymmetric, since $A \subseteq A$, $A \subseteq B$ and $B \subseteq C$ imply $A \subseteq C$ and $A \subseteq B$ and $B \subseteq A$ imply $A = B$. If there were a set of all sets then $R = \{(A, B) : A \subseteq B\}$ would be a partial order on it, but we've already seen that there can be no such set. The construction above does work for any set of sets though and was in fact one of our examples in the section on partial orders. It just doesn't make sense to apply it to the set of

sets because there is no such thing.

Inclusion doesn't really provide a notion of size which agrees with our intuitive notion of size though. If x, y and z are distinct then we would like to be able to say that the set $\{x\}$ is strictly smaller than the set $\{y, z\}$, even though it's not one of its proper subsets. The obvious way to do this is to count the members, but we would like a definition which also works for infinite sets. The standard way to do this is to define the notion of size in terms of the existence of injective functions. There is an injective function from $\{x\}$ to $\{y, z\}$. In fact there are two, $\{(x, y)\}$ and $\{(x, z)\}$. There is no injective function from $\{y, z\}$ to $\{x\}$.

Motivated by this example, we say that A is no larger than B if there is an injective function from A to B. We say that A is of the same size as B if A is no larger than B and B is no larger than A. We say that A is strictly smaller than B if A is no larger than B and there is not an injective function from B to A.

One case where we know there is an injective function is when A is a subset of B. In this case I_A is an injective function from A to B. So we get the rather unsurprising result that a subset is no larger than the set of which it's a subset.

Unwrapping the definitions, A is of the same size as B if there is an injective function from A to B and an injective function from B to A. This is certainly true if there is a bijective function from A to B. If F is such a function then F is an injective function from A to B and F^{-1} is an injective function from B to A. For finite sets this is the only way for two sets to have the same size. In other words, if A and B are finite sets and F is an injective function from A to B and G is an injective function from B to A then F and G are bijective, although it's not necessarily the case that $G = F^{-1}$.

For infinite sets the situation is more complicated. It's possible for there to be an injective but not bijective function function F from A to B and an injective but not bijective function G from B to A. In fact it's possible to give a simple example. Let $A = N$ and $B = N$ and let $F = G = \{(x, y) \in N \times N : y = x + 1\}$. This is the increment function. It's injective because for any natural number x there is a natural number y such that $y = x + 1$. It's not surjective because there is a natural number y for which there is no natural number x with $y = x + 1$. $y = 0$ is such a number,

and is in fact the only such number. So we can certainly have an injective but not bijective function function F from A to B and an injective but not bijective function G from B to A. There is however a useful theorem, the Schröder-Bernstein theorem, which says that in such a case there is always some bijective function H from A to B. It follows that A and B are of the same size if and only if there is a bijective function from A to B. This isn't the definition of having the same size but it is equivalent to that definition as a consequence of the Schröder-Bernstein theorem.

One other difference between finite and infinite sets, or perhaps more accurately the same difference from a different point of view, is that an infinite set can be of the same size as one of its proper subsets. For example the set of natural numbers and the set of positive integers are of the same size since the increment function is a bijective function from one to the other, but it is a proper subset.

The notion of size based on injective functions is called cardinality. Sets of the same size are said to have the same cardinality and a set which is strictly smaller than another set is said to have a lower cardinality than it.

Cardinality behaves somewhat like a partial order. It is reflexive in the sense that any set A is of the same size as itself, since the identity function is injective. It is transitive in the sense that if A is no larger than B and B is no larger than C then A is no larger than C. This is a consequence of the fact that the composition of an injective function from A to B with an injective function from B to C is an injective function from A to C. It is sort of antisymmetric in the sense that if A is no larger than B and B is no larger than A then A is of the same size as B. For true antisymmetry this would have to imply that $A = B$ rather than merely that they're of the same size. Of course "is no larger than" isn't a true relation because there is no set of sets for it to be a relation on. When we restrict it to subsets of a given set it does become a relation though.

The distinction between a partial order on a set and a total order on a set is that the latter has the additional requirement that for all x and y either (x, y) or (y, x) is a member. Even though there is no set of sets we can still ask whether for all sets A and B it is true that A is no larger than B or B is no larger than A. The answer is yes if at least one of the sets is finite. For infinite sets the answer, based on the axioms presented so far, is maybe. It is not possible to prove this but it is also not possible to disprove it.

Diagonalisation

Suppose A is a set and B is PA, i.e. the set of subsets of A. Let F be the set of ordered pairs of the form $(x, \{x\})$ for x in A. It's easy to check that F is both left total and right unique so it is a function. It's also easy to see that it is left unique and so is an injective function. It is not right total though because there is no $w \in A$ such that $(w, \emptyset) \in F$. So F is not surjective. Since there is an injective function from A to B we conclude that A is no larger than B.

Is there some other function G from A to B which is surjective? If there were then we could form the set

$$C = \{x \in A : \exists D \in B : (x, D) \in G \wedge [\neg x \in D]\}.$$

G was assumed to be surjective so there is a $y \in A$ such that $(y, C) \in G$. Either y is a member of C or it isn't. If y is a member of C then there is a D such that $(y, D) \in G$ and $\neg y \in D$. Now y is a member of C but not of D so C and D are not equal. But (y, C) and (y, D) belong to G, which is right unique, so C must be equal to D. So the assumption that y is a member of C leads to a contradiction. Suppose then that y is not a member of C. Then there is a set D such that $(y, G) \in G$ and $\neg y \in D$. Indeed $D = C$ has both these properties. But then the definition of C tells us that $y \in C$, which contradicts our assumption that y is not a member of C. So y is neither a member of C nor not a member of C. The only way to resolve this paradox is that the set C does not in fact exist. But the existence of C follows from that of G by Separation, so G does not exist either. In other words there is no surjective function from A to B.

The argument above is known as the Cantor diagonalisation argument.

Using the Schröder-Bernstein theorem one can sharpen this result somewhat. We've already seen that there is an injective function from A to B. If there were an injective function from B to A then the Schröder-Bernstein theorem would imply the existence of a bijective function from A to B and hence a surjective function from A to B. We've just seen that there is no such function so there can't be an injective function from A to B. In other words A is strictly smaller than B.

For finite sets the size is determined by the number of members and if A has m members then B has 2^m members. We can conclude that $m < 2^m$

for all m. This is indeed true, but hardly surprising. For infinite sets we get a more interesting conclusion. If A is an infinite set then PA is strictly larger than A, so there are infinite sets which are not of the same size. We don't have to stop there though. PPA is strictly larger than PA and $PPPA$ is strictly larger than PPA. There is no limit on the number of infinite sets of different sizes we can construct.

Incidentally, this gives us a different proof of the fact that there is no set of all sets. If there were then every subset of it would be a set and hence a member of itself so the set of sets would contain its own power set. It would therefore have a power set which is no larger than itself, in contradiction to what we've just proved.

Countable sets

We've just seen that there are infinite sets of different sizes. We want a notion of sets which are not too infinite. A set is said to be countable if it is no larger than the set of natural numbers.

There are unfortunately two conflicting terminologies in use. One convention is the one given above. The other defines the countable sets to be those which are of the same size as the set of natural numbers. Under the convention I'm using finite sets are countable. Under the other convention they are not. Both conventions agree on calling a set uncountable if the set of natural numbers is strictly smaller than it. The alternative convention has the rather unfortunate property that "uncountable" and "not countable" are not synonyms. Finite sets are neither countable nor uncountable in this convention. Perhaps more importantly, the condition that a set is no larger than the set of natural numbers arises more frequently in both the hypotheses and conclusions of theorems than the condition that a set if of the same size as the set of natural numbers so it's much more convenient to have a short name for the former condition than for the latter.

There are two unfortunate consequences of this terminological confusion. First, if you read the word countable by itself somewhere other than these notes you can't be sure what the authors mean unless they have explicitly said which convention they follow. Second, if you write the word countable by itself and don't specify which convention you follow then no one can be sure what you mean. The standard way to avoid the second problem

is to refer to sets which are countable according to the definition at the beginning of this section as "at most countable" and to refer to sets which are countable according to the other convention as "countably infinite". This involves some redundancy. According to the convention of these notes the words "at most" in "at most countable" are redundant. According to the other convention the word "infinite" in "countably infinite" is redundant.

Properties of countable sets

Using the convention described above, finite sets are countable. This is reasonably straightforward to prove by induction on sets. $\emptyset$ satisfies all the requirements to be an injective function from $\emptyset$ to N so $\emptyset$ is no larger than N and is therefore countable. Suppose A is a countable set. Then there is an injective function from A to N. If x is a member of A then $A \cup \{x\} = A$ and so F is also an injective function from $A \cup \{x\}$ to N. If x is not a member of A then we can define a set of ordered pairs G whose members are $(x, 0)$ and $(y, m + 1)$ for all (y, m) in F. This G is an injective function from $A \cup \{x\}$ to N. So in either case there is an injective function from $A \cup \{x\}$ to N and so $A \cup \{x\}$ is countable. So $\emptyset$ is countable and if A is countable then so is $A \cup \{x\}$. By induction on sets it follows that all finite sets are countable.

Subsets of countable sets are countable. Suppose that A is a subset of B and B is countable, i.e. there is an injective function G from B to N. Define F to be the subset of ordered pairs in G whose left element is a member of A. Then F is an injective function from A to N, so A is countable. It follows from this that if B is countable and C is a set then both $B \cap C$ and $B \setminus C$ are countable, since they are subsets of B.

The union of two countable sets is countable. To see this, suppose A and B are countable. We've just seen above that $A \setminus B$ is then countable, i.e. that there is an injective function from $A \setminus B$ to N. Let F be such a function. B is countable so there is an injective function G from B to N. Define H to be the set of pairs either of the form $(x, 2 \cdot m)$ where $(x, m) \in F$ or of the form $(y, 2 \cdot m + 1)$ where $(y, m) \in G$. Then H is an injective function from $A \cup B$ to N so $A \cup B$ is countable.

N itself is countable since I_N is an injective function from N to N. Perhaps surprisingly $N \times N$ is also countable. It's possible to write down an injective function from $N \times N$ to N explicitly. Such a function is given by the set

of ordered pairs $((i,j),k)$ where

$$k = (i+j)(i+j+1)/2 + j.$$

The division by two is permissible because $(i+j)(i+j+1)$ is always even, as we can prove by induction.

The function above may appear mysterious but it is easily explained by the following picture.

⋮							
5	20						
4	14	19					
3	9	13	18				
2	5	8	12	17			
1	2	4	7	11	16		
0	0	1	3	6	10	15	
	0	1	2	3	4	5	⋯

The horizontal axis is labelled by the i values, the vertical axis by the j values and the element in the i'th column, j, row, counting from the bottom left and starting at 0, is the corresponding k value. You can see that these numbers are obtained by visiting the pairs in a particular order, working one diagonal at a time and going from the lower right to the upper left within that diagonal. Working out how many points in the grid are visited before the given point gives exactly the expression above. and the fact that this function is injective is simply the fact that this procedure never reuses a natural number. This is visually obvious but rather tedious to prove.

More generally, if A and B are countable then so is $A \times B$. To see this note that in this case there are injective functions F from A to N and G from B to N. Define H to be set of pairs of pairs $((x,y),(m,n))$ such that $(x,m) \in F$ and $(y,n) \in G$. Then H is an injective function from $A \times B$ to $N \times N$. Composing this with the injective function we already have from $N \times N$ to N gives an injective function from $A \times B$ to N, so $A \times B$ is countable.

In particular, if A is countable then so is A^2. Since the Cartesian product of two countable sets is countable it follows that $A^2 \times A$ is countable. There is an injective function from A^3 to $A^2 \times A$ consisting of the ordered pairs of the form $((x,y,z),((x,y),z))$. This function is also surjective, but we won't

need that. The fact that it is injective means, together with the fact we just proved that $A^2 \times A$, implies that A^3 is countable.

Less obviously, if A is countable then so is the set of all lists all of whose items are members of A. This fact is of great importance in the study of formal languages. Since subsets of countable sets are countable and languages are sets of lists of tokens it follows that every language with a countable number of tokens is countable.

Here is a sketch of a proof of the statement above. A is countable so there is an injective function F from A to N. Define a function G from A^* to N^3 as follows. $(w, (x, y, z)) \in G$ if x is the number of elements in the list w, y is the least natural number n such that if v is an item in w and $(v, m) \in F$ then $m < n$, and z is natural number whose base n representation has as it's j'th digit the number k where $(v, k) \in F$ and (v) is the j'th item in the list. This G is an injective function. There is an injective function H from N^3 to N. Then $H \circ G$ is an injective functions from A^* to N, so A^* is countable.

Uncountable sets

It's easy to produce uncountable sets. PN is uncountable. If it were countable then there would be an injective function from PN to N but we've already seen that there can be no such function.

Let A be the set of arithmetic sets, i.e. subsets of N for which there is a Boolean expression in our language for arithmetic which is a necessary and sufficient condition for membership in the set. Choose some encoding of that language into N. Consider those pairs (B, x) with $B \in A$ and $x \in N$ such that x is the natural number which encodes a Boolean expression characterising membership in B. The set of such pairs is an injective function from A to N, so A is countable.

A is not PN because A is countable and PN is uncountable. A is a subset of PN so there must therefore be a member of PN which is not a member of A. In other words there is a subset of N which is not arithmetic. We've already seen an example, without a proof, of such a set, namely the set of encodings of true statements. That's a hard theorem though while the proof above, while it doesn't provide any examples, is quite easy.

A similar argument shows that there is a language which has no phrase

structure grammar. We choose as our set of tokens a non-empty countable set. Let A be the set of lists of tokens. A is then countable. We can say a bit more than that though. Since the set of tokens is non-empty we can choose one and look at the set of lists using only that token. This, as we discussed, is essentially a copy of N. So N is no larger than A but A is also no larger than N because it's countable. Therefore A is of the same size as N. It follows that PA, which is the list of languages using only those tokens, is uncountable. Any phrase structure grammar for A is a list of tokens. These tokens belong to the original list of tokens or are tokens like ":", "|", or ";" which belong to our language for describing languages. There are only finitely many of the latter though so the full set of tokens is still countable and therefore the set of phrase structure grammars is countable. There are fewer phrase structure grammars than languages so there is a language without a grammar.

As with arithmetic sets, it is possible to give concrete examples of languages with no phrase structure grammar but the proofs are much harder than the simple counting argument above.

Choice

The axioms we've seen so far are arguably sufficient for all of computer science and are sufficient for some mathematics, including at least elementary arithmetic. There is some standard mathematics for which they are insufficient though.

From now on I'll use N to denote the set of natural numbers. It won't matter what representation of N we use.

Dependent choice

Consider the following two statements.

- Suppose R is a left total relation on a set A. For every $x \in A$ there is a function F from N to A such that $(0, x) \in F$ and if $(n, y) \in F$ and $(n + 1, z) \in F$ then $(y, z) \in R$.
- For every set A and partial order R on A if every chain in A is finite then A contains a maximal element.

The chains and maximal elements in the second statement are understood as being with respect to the partial order R.

The meaning of these statements may not be immediately obvious but the first one, at least, has a fairly nice interpretation in terms non-deterministic computations. We think of A as the state space and x as the initial state. The function F simply tells us which state the computation is in after a given number of steps. The relation R is the one which gives the allowed transitions. The condition that if $(n, y) \in F$ and $(n + 1, z) \in F$ then $(y, z) \in R$ is then the statement that each transition is one of the allowed ones and the condition that $(0, x) \in F$ is of course the condition that that the computation starts in the start state. The condition that R is left total means that for any allowed state there is at least one state to which the system can transition, so the computation is never forced to terminate due to a lack of options. We can therefore give the first statement the interpretation that a non-deterministic computation for which there is at least one transition from each allowed state can run forever.

There are two important conditions under which we can prove the first statement using the axioms we've introduced so far. One is when the computation is deterministic, i.e. when F is not just a left total relation but a function. The other is when the set A of possible states is countable. Those two cases are probably sufficient for computer science.

The second statement is harder to attach an intuitive meaning to but it is in some sense simpler. It's certainly shorter and it doesn't refer to the natural numbers so it makes sense even without the Axiom of Infinity.

If we assume the Axiom of Infinity then the two statements above are in fact equivalent, in the sense that from either we can prove the other. They are independent of our axioms though in the sense that neither of them can be proved from those axioms. If we want them to be true then we'll need to take one of them as an axiom, or choose some other axiom which implies them. I'll choose the second one.

- Dependent Choice: For every set A and partial order R on A if every chain in A is finite then A contains a maximal element.

Zorn's Lemma

If we think of classical mathematics as covering the integral and differential calculus, complex analysis, ordinary and partial differential equations, number theory, differential geometry and classical algebraic geometry then the axioms above are a sufficient foundation. There are some parts of modern mathematics which require something stronger than the Axiom of Dependent Choice though. There are a few different, but equivalent, choices for what this something is but the most popular is the following.

- Zorn's Lemma: For every set A and partial order R on A if every chain in A has an upper bound in A then A contains a maximal element.

Every finite chain has a maximal element and this maximal element is an upper bound so the Axiom of Dependent Choice follows from Zorn's Lemma. Zorn's Lemma does not follow from the Axiom of Dependent Choice though. We can therefore think of Zorn's Lemma as a generalisation of the Axiom of Dependent Choice.

Despite its name, Zorn's Lemma is normally taken as an axiom and so is not a lemma. It's also due to Kuratowski rather than to Zorn. The word lemma in its name dates to a time when it was traditional to take the following as an axiom:

- Choice: Suppose A is a set of non-empty sets such that if $B \in A$ and $C \in A$ then $B = C$ or $B \cap C = \emptyset$. Then there is a set D such that for all $B \in A$ the intersection $B \cap D$ has exactly one member.

D can be thought of as selecting a single member from each member of A. That's why it's called the Axiom of Choice.

Kuratowski showed that the "lemma" is a consequence of this axiom. Zorn stated the axiom also follows from the lemma and promised to prove this, but didn't. It is nonetheless true. The two statements are equivalent, just as the two statements in the previous section were. Which one to take as an axiom and which to prove as a consequence is then just a matter of convenience. The modern approach is to take the "lemma" as an axiom and prove the "axiom" as a theorem.

Banach-Tarski

Unfortunately the Axiom of Choice has some rather unsettling consequences. Perhaps the most counterintuitive of these is the Banach-Tarski Paradox in geometry. Assuming Zermelo-Fraenkel plus the Axiom of Choice one can show that there are sets A_1, A_2, A_3, A_4, A_5, B_1, B_2, B_3, C_1, C_2, C_3, C_4 and C_5 in three dimensional Euclidean space with the following properties.

- B_1, B_2 and B_3 are disjoint balls of radius 1.
- A_1 is congruent to C_1, A_2 is congruent to C_2, A_3 is congruent to C_3, A_4 is congruent to C_4, and A_5 is congruent to C_5.
- A_1, A_2, A_3, A_4 and A_5 are disjoint and their union is $B_1 \cup B_2$.
- C_1, C_2, C_3, C_4 and C_5 are disjoint and their union is B_3.

In other words, we can take a ball, split it into five pieces, move those pieces via a rigid motion, i.e. a combination of translations, reflections and rotations, and reassemble them to form two balls of the same radius as the original one.

Most mathematicians are not particular bothered by paradoxes like the one above. In their view it shows that it's possible to find really weird bounded subsets of Euclidean space, weird enough that one can't attach a notion of volume to them in any consistent way, but not as a problem with set theory. Some mathematicians reject the Axiom of Choice entirely. Others accept only weaker versions, like the Axiom of Dependent choice, which do not imply the existence of the paradoxical sets appearing in Banach-Tarski theorem.

Additional axioms

Foundation

The following was not part of Zermelo set theory but is often taken as an axiom.

- Foundation: Every non-empty set has a member which is disjoint

from it, i.e. shares no members with it. Formally

$$[\forall A.[[\exists B.B \in A] \supset [\exists C \in A : [A \cap C] = \emptyset]]].$$

I'm not sure I've ever seen anyone present an argument that this statement is true, as opposed to simply convenient.

Some programming languages provide a set data type, which generally means a finite set data type, natively while there are library implementations in some others. Most of those do not appear to satisfy the Axiom of Foundation, which makes it hard to argue that this axiom reflects people's intuitive understanding of sets, even when restricted to finite sets. Even the arguments that Foundation is convenient are somewhat suspect since it is generally assumed by mathematicians but never really used by them.

It is at least safe to assume it, in the sense that if it is possible to prove a contradiction using this axiom then it is also possible to prove one without it.

Extensionality, again

There is another, stronger, form of the Axiom of Extensionality.

- Extensionality (stronger version): Suppose every member of A is a member of B and vice versa. Then $A = B$.

Formally,

$$[\forall A.[\forall B.[[\forall x.[[[x \in A] \supset [x \in B]] \wedge [[x \in B] \supset [x \in A]]]]] \supset [A = B]]].$$

The difference between this version and the previous version is that one began with the words "Suppose A and B are sets" and the formal version had some additional conditions expressing that assumption. So the strong version of the axiom implies that if A has no members then A is the empty set. This might seem innocuous but it means that for all A, A is a set, which is an additional assumption we haven't made previously. Assuming this axiom, whenever we see a statement of the form $x \in A$ not only must A be a set but so must x. I've been deliberately vague about what can be a member of a set but with this version of Extensionality the answer is that the only things which can be members of sets are sets.

Can we have a set of natural numbers? That is compatible with the strong version of Extensionality because we've implemented natural numbers as sets. We can also implement integers, rational numbers, real numbers and complex numbers as sets. The usual way to implement the integers, for example, is as equivalence classes of ordered pairs of natural numbers, where the equivalence relation is the set of pairs of pairs $((v, w), (x, y))$ for which $v + y = w + x$.

In general this version of Extensionality is compatible with modern mathematics. Whether it's a good idea or not is another question. It forces us to implement everything as a set. The fact that we can do this doesn't necessarily mean we want to be forced to.

Assuming this version of Extensionality, if we start with a set then all of its members, if it has any, are sets. All of their members, if they have any, are sets. The same applies to their members. Starting from a set, choosing one of its members, then one of its members, etc. we get a sequence of sets which is either infinite or terminates with the empty set. If we assume the Axiom of Foundation and the Axiom of Replacement as well then it cannot give an infinite sequence and so must terminate with the empty set. So in some sense all sets are built from only the empty set.

Zermelo-Fraenkel

The most common choice of axioms for set theory is

- the strong version of Extensionality,
- Elementary Sets, without assuming the existence of the empty set, which we can get from Separation and Infinity, or if you really take a first order logic with existential presuppositions seriously, from the mere fact that it has a name.
- Separation,
- Power Set,
- Union,
- Infinity, but the version corresponding to implementing the natural numbers as von Neumann ordinals,

- Replacement, in the version which constructs the range rather than the function,
- Foundation,
- Zorn's Lemma, or, equivalently, the Axiom of Choice.

The version without the Axiom of Choice is known as Zermelo-Fraenkel, or just ZF. The version with Choice is just called Zermelo-Fraenkel with the Axiom of Choice, or ZFC. Zermelo would have hated this terminology. He felt that Fraenkel had vandalised his theory by adding some of the axioms above and wouldn't have wanted to have his name associated with ZF or ZFC.

Most introductions to set theory start with examples from outside mathematics, e.g. the set of students in a particular class, the set of people in a building, etc. In view of the comments on the strong version of Extensionality none of these things are sets in Zermelo-Fraenkel, with or without the Axiom of Choice. In fact Zermelo-Fraenkel set theory is aggressively hostile to possible applications outside of mathematics.

A version of set theory sufficient for large parts of mathematics and nearly all of computer science, and less openly hostile to other applications, would be

- the original version of Extensionality
- Elementary Sets
- Separation
- Power Set
- Union
- Infinity, in some version
- Replacement, in some version
- Dependent Choice

Even Replacement isn't really needed if you take a strong enough version of Infinity. You don't really need any form of Choice for computer science but Dependent Choice is useful. The axioms above, together with Dependent Choice, are also sufficient for nearly all of classical mathematics. For some

parts of modern mathematics the full Axiom of Choice is useful, but then you have to accept paradoxes like Banach-Tarski. Foundation is probably best forgotten.

Graph theory

The word "graph" has multiple, unrelated, meanings in mathematics. What we're concerned with here is not graphs of functions but rather graphs as they are understood in the field known, appropriately enough, as graph theory. A graph is a set of vertices and edges. The edges connect vertices. There are, in fact, two different ways to make this notion precise, depending on whether we regard the connections between vertices to have a direction or not. These are called directed graphs and undirected graphs.

Examples

Before giving definitions, it may be helpful to consider examples of each.

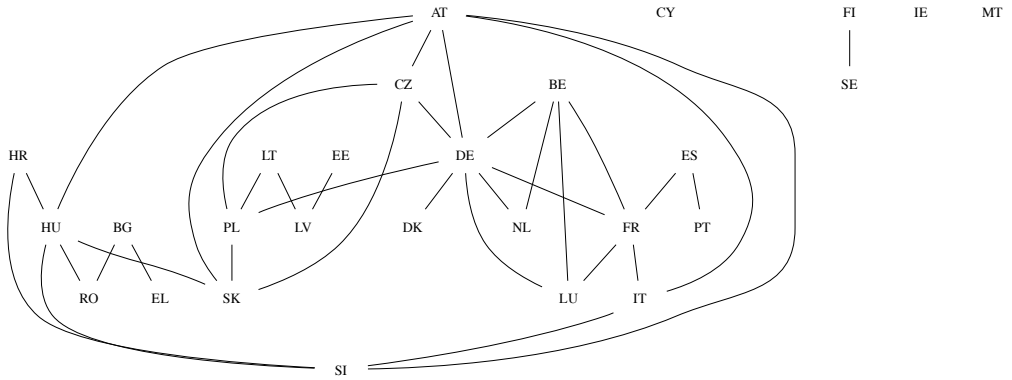


Figure 40: An undirected graph

The first example is of land borders within the EU. Vertices are countries and edges are land borders between them. This is an undirected graph, because there is no preferred direction for border crossings.

The second example has as its vertices the substrings of the word mathematics which are themselves words. There is an edge from one word to

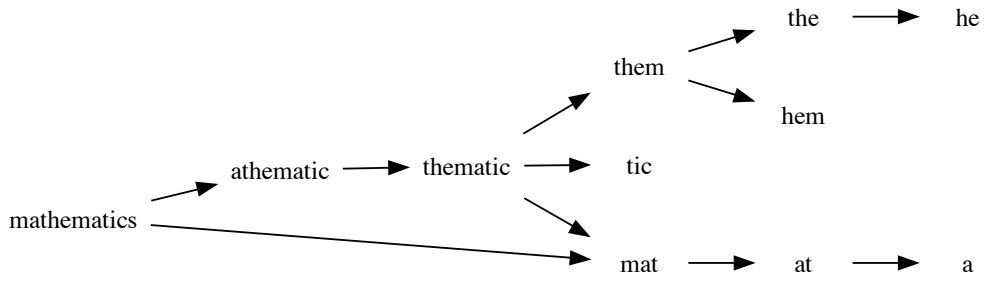


Figure 41: A directed graph

another if the second occurs as a proper substring of the first without any other word appearing in between. This is a directed graph because the "is a proper substring of" relation is not symmetric. You might notice that this graph is very nearly a tree. It only fails to be because the word mat appears twice in mathematics. The first occurrence is a proper substring of mathematics but not a proper substring of any proper substring which is a word. The second occurrence is a proper substring of the word thematic.

Note that the graph is defined by which vertices are connected by an edge, not by its visual representation in a particular diagram. There are edges which cross in our directed graph example. This could have been avoided by rearranging the positions of some vertices and edges but the crossings are in any case just artifacts of the particular visual representation, not features of the graph. A graph which can be drawn in the plane without edge crossings is called planar. So the EU border graph is planar, even though this particular diagram has edge crossings. Not all graphs are planar though. Our third example, with seven vertices and an edge between each pair of vertices, is not. Proving that a graph isn't planar is not straightforward though, since the presence of edge crossings in some particular diagram doesn't really tell you anything. The only way to prove this is to prove that all planar graphs have some property which all planar graphs have and then show that this graph doesn't have it.

You can find a number of other examples of graphs in earlier chapters. All trees are graphs. Also, all of our state diagrams for idealised machines are graphs, provided we make one change, which is described below.

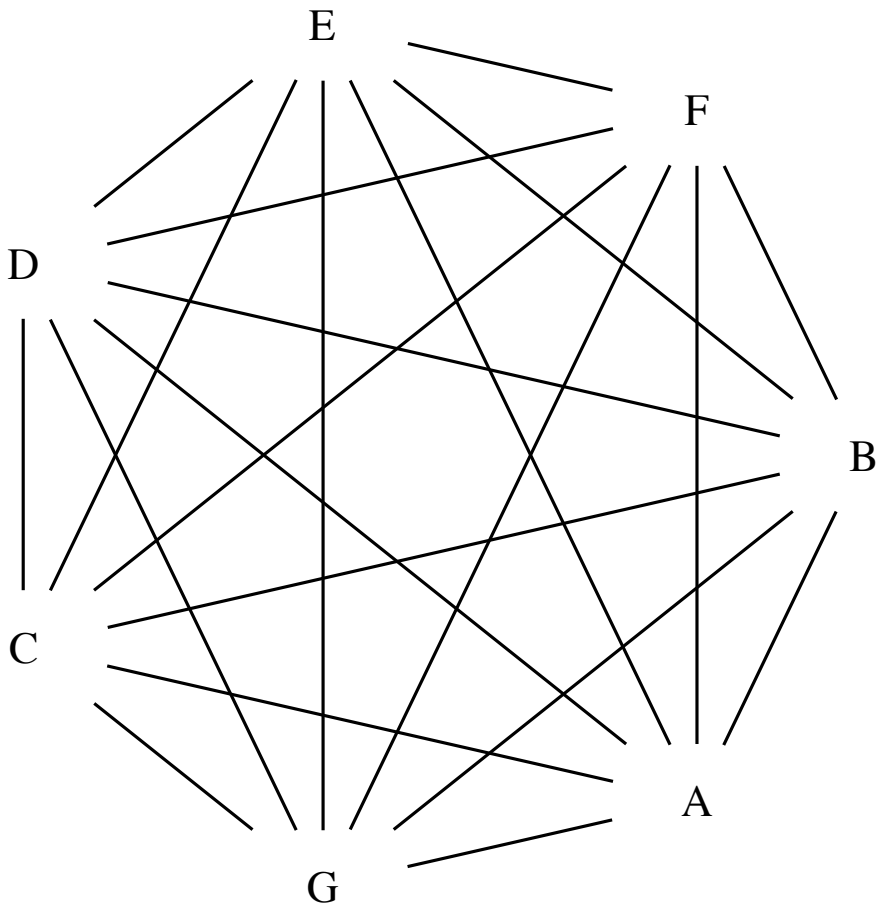


Figure 42: Another undirected graph

Different notions of a graph

Graph theory is really a collection of closely related theories which differ in some details, depending on a few basic choices:

- Are our graphs directed or undirected?
- How many vertices and edges do we allow? Finitely many? Countably many, uncountably many?
- Are self-loops, i.e. edges connecting a vertex to itself, allowed?
- Can there be more than one edge between a pair of vertices?

For different applications different combinations of these are useful. We don't have time to cover all of them though and so will have to make some choices.

I'll take graphs to be directed unless otherwise specified. Most of the graphs we've encountered are best thought of as directed graphs. State transitions in an idealised machine often go in one direction only. Undirected trees are sometimes useful but most of our trees, e.g. abstract syntax trees or trees representing possible paths for a non-deterministic computation, have a natural direction to their edges, from parent to child. There is no real loss of generality in considering graphs to be directed. We can always think of an undirected graph as a special case of a directed graph where for each edge from on vertex to another there is a corresponding edge in the reverse direction. It's linguistically a bit unfortunate that undirected graphs are directed graphs but a lot of mathematical terminology has similar properties. The only real disadvantage of this point of view is that you have to be careful reading works which deal only with undirected graphs. They will use the word edge to refer to what we're considering a pair of edges. Later we'll consider Eulerian paths in an undirected graph, for example. In a text devoted solely to undirected graphs these would usually be described as traversing each edge exactly once. If you're considering undirected graphs as directed graphs then you need to modify this to say that a path is Eulerian if from each pair of oppositely directed edges it traverses one edge exactly once and the other not at all. To avoid clutter in diagrams, whenever we have an undirected graph I will show a single edge without arrows rather than a pair with arrows, as I did in the first and third examples above. That

convention is limited to undirected graphs however. For graphs which are not undirected I will show both edges where there are two.

I will allow infinite graphs, but will restrict attention to finite graphs wherever that's convenient. The graphs associated to finite state automata, abstract syntax trees, and computational paths of processes guaranteed to terminate are all finite. The tree of computational paths of a non-terminating process is infinite.

I will allow self loops in the definition, because they arise naturally in graphs for finite state automata. For some theorems though it will be necessary to add a hypothesis that the graph has no self loops.

I'll exclude the possibility of having more than one edge from one vertex to another. This means that for a finite state automaton where there is more than one input token which causes a given transition we need to list all of those in the label on a single edge rather than having multiple edges each labelled by a different token. Note that the restriction is only on multiple edges from one vertex to another. We are allowed to have two edges between a pair of edges as long as they go in different directions.

The choices above are motivated mainly by applications to the theory of computation. Graph theorists tend to make a different set of choices, preferring undirected finite graphs with no self loops.

Definition

With the conventions chosen above we can define a graph as a set, the set of vertices, and a relation on that set, the set of order pairs of vertices for which there is an edge connecting the left component of the pair to the right component. From this point of view graph theory is just the study of relations on sets, but the questions we ask when considering such a relation as a graph are different from the ones we normally ask about relations. It is sometimes helpful though to recast problems about relations as problems about graphs, to take advantage of our spatial intuition.

With this definition a graph is undirected if and only if it is symmetric, i.e. if and only if (x, y) belongs to the edge relation whenever (y, x) does. Self loops are just pairs of the form (x, x).

If we allowed multiple edges from one vertex to another we would have to adopt a different set of definitions. The usual way to to this is to have two sets, for vertices and relations, and two relations, each of which applies to a vertex and an edge. The first is the "is the initial vertex of" relation and the second is the "is the final vertex of" relation.

Ways to describe finite graphs

All the examples so far have been given via diagrams. This works well for human viewers and small graphs, but becomes unwieldy for larger graphs or for machine processing. There are several alternative ways to describe finite graphs. As an example, consider the graph whose diagram has vertices labelled a to e and edges labelled 1 to 6.

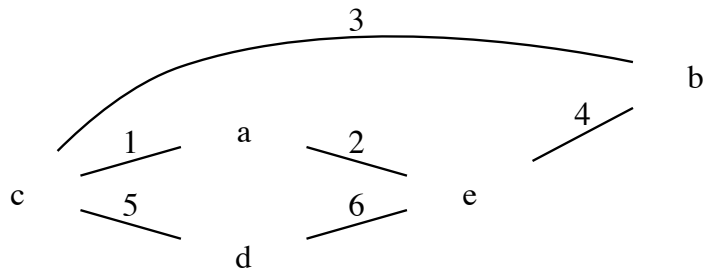


Figure 43: An undirected graph with labelled edges

One way to describe this is with what's called an incidence table, as shown below

	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	0	1	1	0	0
c	1	0	1	0	1	0
d	0	0	0	0	1	1
e	0	1	0	1	0	1

There is a row for each vertex and a column for each edge. There is a 1 in the row corresponding to a vertex and the column corresponding to an

edge if that vertex is an endpoint of that edge, and a 0 otherwise. If we remove the row and column labels then we get an incidence matrix:

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

There isn't a particularly good analogue of this for directed graphs. Sometimes people use an incidence matrix with a -1 entry for the initial endpoint and 1 for the final endpoint.

An alternative way to describe a graph is with an adjacency table. This has a row and a column for each vertex. There is a 1 in a row and column if the graph has an edge from the vertex corresponding to that row to the vertex corresponding to that column and a 0 otherwise.

	a	b	c	c	e
a	0	0	1	0	1
b	0	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	1	1	0	1	0

Again, we can remove the labels to get a matrix, the adjacency matrix:

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

This is a symmetric matrix, reflecting the fact that our graph is undirected. It has 0's along the main diagonal, reflecting the fact that the graph has no loops.

This representation works well in the case of graphs which are not undirected as well. For our earlier example of a directed graph, the one with

substrings of the word mathematics, the adjacency matrix is

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

For each representation, we need an order relation to determine the order of the rows and columns. For the adjacency matrix we need an ordering of the vertices. For the incidence matrix we need that and an ordering of the edges. Different choice of order relation will require permuting the rows and columns of the matrices. In the particular case above I chose to order the strings first by length and then lexicographically within those of each possible length. A more traditional choice for ordering words would be just to use lexicographic ordering, but this would disguise an important property of our graph: the fact that it is possible to order the vertices in such a way that all edges go from vertices earlier in the order to vertices later in the ordering. Graphs with this property are called directed acyclic graphs. The come up in a variety of contexts. With such an ordering the adjacency matrix is lower triangular.

As often happens there differing conventions here. For directed graphs I've chosen to make the rows of the adjacency matrix correspond to initial endpoints of an edges and make the columns correspond to the terminal endpoints. Roughly half the world seems to use that convention and half uses the reverse convention. The effect of changing conventions is to transpose the matrices.

Bipartite graphs, complete graphs, colouring

A graph is called complete if it has no self loops but otherwise has an edge from each vertex to each other vertex. Complete graphs are undirected.

The graph I gave earlier as an example of a non-planar graph is complete. The adjacency matrix of a complete graph looks like an identity matrix with the 1's and 0's reversed. A complete graph with n vertices, known as a K_n. Our example graph is therefore a K_7.

A graph is called bipartite if the set of vertices can be partitioned into two subsets, such that all the edges connect a vertex from one subset to a vertex from the other. An example is the graph above with labelled edges. The two subsets of vertices are $\{a, b, d\}$ and $\{c, e\}$, in the labeling from that diagram. This bipartite graph has the property that for every vertex in the first subset and every vertex in the second there is an edge connecting them. That is not a requirement of the definition. A bipartite graph with p vertices in one set and q in the other is called a $K_{p,q}$. These graphs are often referred to as "complete bipartite" graphs. This terminology is unfortunate, because these graphs are not in fact complete graphs for $p > 1$ or $q > 1$, so I won't use it.

Another example is given in the figure after. In this case it's easy to see that the graph is bipartite because every edge connects an even numbered vertex to an odd numbered one.

Bipartite graphs arise in a variety of contexts. The one above is related to the finite projective plane of order 2. The even numbered vertices correspond to points and the odd numbered vertices correspond to lines. There is an edge joining two vertices if and only if they correspond to a point and a line through that point.

More generally we can consider graphs whose set of vertices can be partitioned into k disjoint subsets such that no edge connects two vertices in the same subset. Such a partition is called a k-colouring of the graph. A bipartite graph is then one with a 2-colouring. It's not terribly difficult to show that every finite planar graph has a 5-colouring, and indeed to give reasonably efficient algorithms for finding a 5-colouring for a given graph. It's considerably more difficult to show that every planar graph has a 4-colouring. There are planar graphs which can't be 3-coloured. Indeed K_4 is such a graph. Deciding whether or not a graph has a 3-colouring is a hard computational problem. Earlier I mentioned that K_7 is not planar. In general K_n can't be coloured with fewer than n colours so it follows from the Four Colour Theorem that K_n is not planar for $n > 4$.

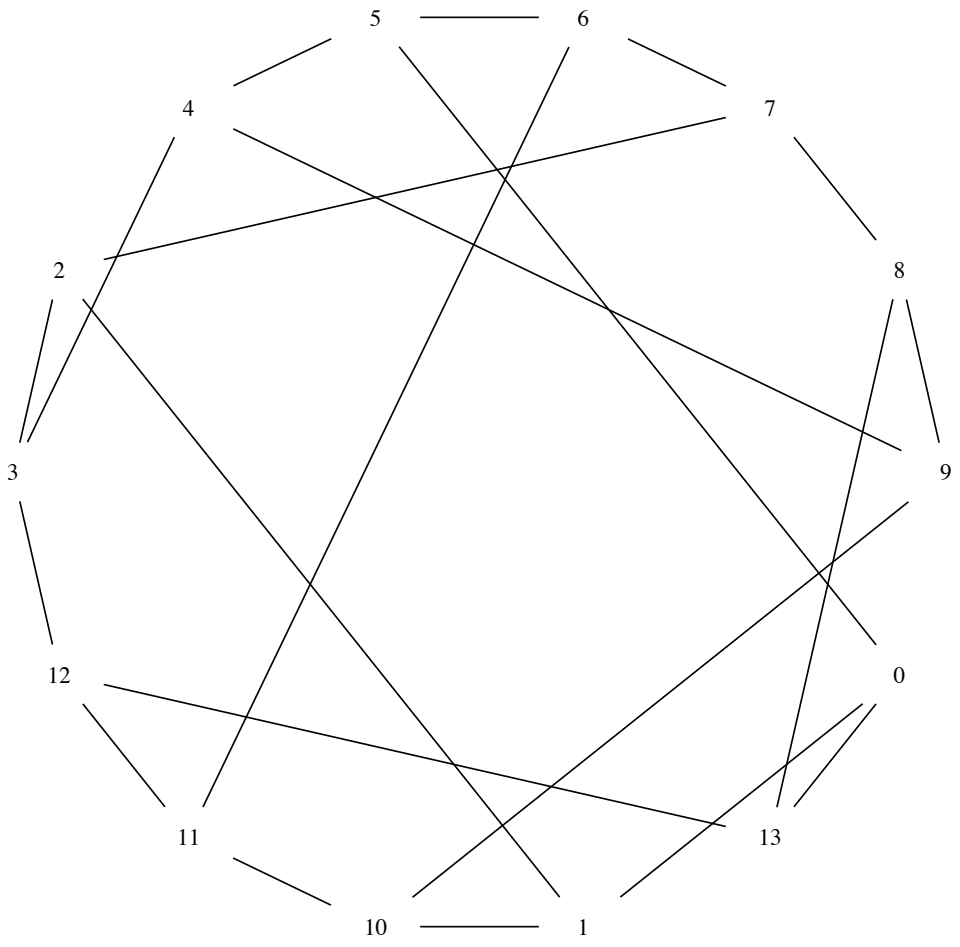


Figure 44: A bipartite graph

Homomorphisms

Suppose we have two graphs, one with vertex set V and edge relation E and the other with vertex set W and edge relation F. A function g from V to W is called a graph homomorphism $(g(a), g(b)) \in F$ whenever $(a, b) \in E$.

Several concepts we've already defined can be expressed in terms of homomorphisms. For example, a graph E is bipartite if and only if there is a homomorphism g from it to a complete graph with two vertices, which we'll call u and v, since we can partition the vertices E into those for which $g(a) = u$ and those for which $g(a) = v$. The definition of a homomorphism and the fact that complete graphs have no self-loops imply that there are no edges between a vertex in the first set and a vertex in the second set. More generally, a graph has an m-colouring if and only if there is a homomorphism from it to a complete graph with m vertices.

A graph homomorphism is called a graph isomorphism if it is a bijective function and its inverse is also a homomorphism. In other g from V to W is an isomorphism if is bijective and $(g(a), g(b)) \in F$ whenever $(a, b) \in E$ and vice versa. In the special case $W = V$ and $F = E$ it's called an automorphism.

There is rarely much point in distinguishing between isomorphic graphs, and people often implicitly treat isomorphic graphs as equal. For example, people talk of K_n as the complete graph with n vertices. Technically, there is such a graph for each set with n elements, but for purposes of graph theory they all behave the same, and so we speak as if there were only one.

An easy way to describe isomorphism, at least for finite graphs, is that two graphs are isomorphic if and only if their vertices can be ordered in such a way that they have the same adjacency matrix. Or, if we don't want to disturb an ordering that we may already have given the vertices, they are are isomorphic if and only if one adjacency matrix can be converted into the other by applying a permutation to the rows and applying the same permutation to its columns.

Every graph has at least one automorphism, corresponding to the identity function, but even small graphs may have many more. K_n has $n!$ automorphisms, since any bijective function from the set of vertices to itself will be an automorphism. $K_{p,q}$ has $p! \cdot q!$ automorphisms, unless $p = q$, in which

case there are twice as many, because in that case we can not only permute each of the two subsets into which the vertices have been partitioned but can also swap the two subsets. The graph we saw earlier, with vertices labelled 0 to 13, has 336 automorphisms. Our first two examples of graphs, by contrast, have only the identity automorphism.

Subgraphs, degrees

Suppose we have a graph with vertex set V and edge relation E. If W is a subset of V and F is a subset of the restriction of E to F then we say that the with vertex set W and edge relation F is a subgraph of the one with vertex set V and edge relation E. Note that in cases where two vertices x and y in W are connected by an edge in F they are required to be connected by an edge in E, but not vice versa. We could, for example, obtain a subgraph by keeping all the vertices and removing all the edges, although this wouldn't be particularly interesting. For a slightly more interesting example, consider the graph with 14 vertices considered earlier. It can be considered as a subgraph of $K_{7,7}$, since we could add further edges between each even numbered vertex and each odd numbered vertex.

Like many concepts in graph theory, the notion of a subgraph is related to graph homomorphisms. If V is a subgraph of W then the inclusion function from V to W, i.e. the one defined by $g(a) = a$ for all $a \in V$, is a graph homomorphism.

The in-degree of a vertex in a graph is the number of edges from that vertex while the out-degree is the number of edges to that vertex. In undirected graphs these two numbers must be the same and are just called the degree of the vertex. Corresponding vertices in isomorphic graphs have the same in-degrees and have the same out-degree. This can be used to show that a pair of graphs are not isomorphic, by showing that the number of vertices with a given in and out degree differ between the two graphs.

Degrees are generally only useful when they're finite. This is certainly the case for finite graphs but it is possible to have an infinite graph where all the vertices have finite degree.

An undirected graph where all vertices have the same degree is called regular. Complete graphs are always regular. A $K_{p,q}$ is regular if and only if

$p = q$. The EU borders graph considered earlier is very far from regular. There are some vertices, e.g. Ireland, with degree 0 while Germany has degree 8. An example of a regular graph which is not a K_n or $K_{p,p}$ can be found in the accompanying figure, where each vertex has degree 5.

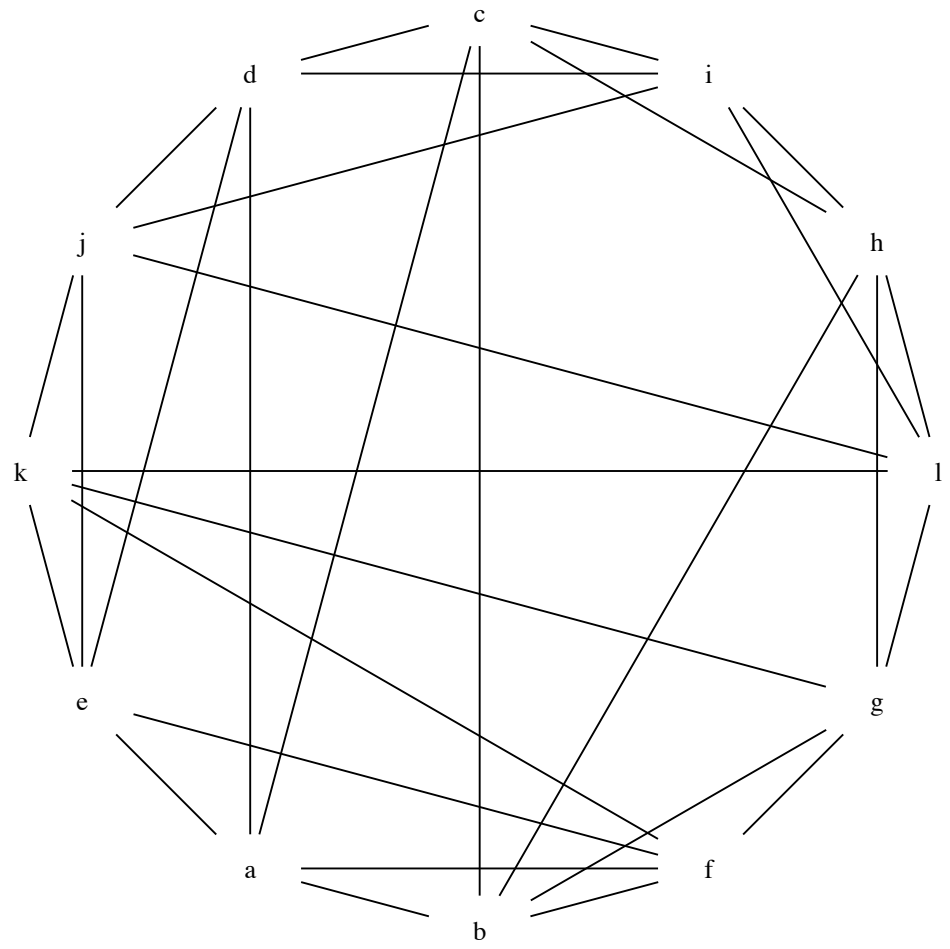


Figure 45: A regular graph

Each edge goes from on vertex to another. If we group the edges by their initial endpoints then the number for each vertex is its out-degree, so the number of edges is equal to the sum of the out-degrees of the vertices. Similarly, if we group them by their final endpoints then we see that the number

of edges is equal to the sum of the in-degrees of the vertices. The sum of the in-degrees is therefore equal to the sum of the out-degrees.

For an undirected graph the in-degrees and out-degrees are the same, so we can just say that the sum of the degrees is equal to the number of edges. We have to be careful here though, because edges occur in pairs an our convention is to draw only one of each pair. The sum of the degrees is therefore twice the number of edges visible in the diagram. This is always an even number so we obtain the useful result that the sum of the degrees of the vertices in an undirected graph is always and even number, and the corollary that the number of vertices of odd degree is even.

Walks, trails, paths, etc.

A walk in a graph is a list of edges where the final endpoint of each edge, other than the last, is the initial point of the next one. Of course for an undirected graph we don't have to worry about which vertex is the initial vertex and which is the final vertex of an edge, since there is always another edge with the opposite orientation. You can check that the edges (a,b), (b,c) (c,d), (d,e), (e,f), (f,g), (g,h), (h,i), (i,j), (j,k), (k,l) form a path of length 11. It's more efficient to list the vertices in order than to list the edges though to avoid listing vertices twice, once as the initial endpoint of an edge and once as the final endpoint. The walk above would then be given by the list of vertices $(a,b,c,d,e,f,g,h,i,j,k,l)$. Note that the number of vertices in such a list is always one greater than the number of edges.

Walks are also describable in terms of graph homomorphisms. Consider the directed graph whose vertices are the natural numbers 0, 1, ..., m and whose edges connect each of these numbers, except the last, to its successor. Specifying a walk of length m in a graph is equivalent to specifying a homomorphism from from the graph just described to it.

A walk in an undirected graph, like the one above, is called a trail if at most one from each pair of edges appears and is called a path if each edge appears at most once. The walk above is both a trail and a path. It can be extended further as a trail but not as a path. We could, for example, extend the path further by adding the edge (l,j) to get a trail of length 12, but this would not be a path since the vertex j would appear twice. In fact there can't be a trail of length 12 in this graph because the number of vertices

appearing in a trail is one greater than the length and this graph only has 12 vertices.

A walk is called closed if it starts and ends with the same vertex. The walk above is not closed, but it can be extended to a closed walk, which visits the vertices in the order given by the following list: $(a, b, c, d, e, f, g, h, i, j, k, l, j, e, k, g, l, h, c, i, d, a)$. This is a closed path of length 21. It is in fact a trail. Closed trails are called circuits.

How long could a circuit in this graph be? The number of edges is the sum of the degrees of the vertices and there are twelve vertices, each of degree 5, so there are 60 edges, or 30 pairs of edges, so no trail could possibly have length greater than 30. In fact we can't even have one that long. The number times a vertex appears as the initial vertex of an edge in a circuit must be equal to the number times it appears as a final vertex and there are only five pairs of edges for each vertex so the we can't have more than two incoming and two outgoing edges appearing in a circuit. With 12 vertices there therefore can't be more than 24 edges.

Suppose a non-empty undirected graph has all vertices of degree at least 2. Then it has a simple circuit. We can see this as follows. Given any vertex v of degree at least two there is a pair of distinct edges through v. Taking one and then the other gives a path length two passing through v. Consider the set of paths through v. The length of such a path is at most the number of vertices in the graph. There is therefore a longest such path. The final point of the path has degree at least two and only one of the edges it traverses is in the path, since the path has no repeated vertices. Adding that edge gives a longer walk, but it can't give a longer path, since we've already chosen one of maximal length. The other vertex of the edge we've added must then be one of the vertices already in the path. Following from that vertex along the path and then back along the edge we've just added gives a simple circuit.

Another case in which we know there is a non-trivial simple circuit is when there are two vertices connected by distinct paths. We can get a closed path by following one path in the forward direction and the other in the reverse direction, but that walk need not be a circuit, let alone a simple circuit. We can, however, look at the first vertex where the two paths diverge and the first vertex after that where they come together again. If we look only at the parts of the paths between those two vertices then we can still follow

one in the forward direction and the other in the reverse direction. This time the resulting closed walk will be a simple circuit.

A trail or circuit is called Eulerian if exactly one from each pair of edges appears. Our example graph has no Eulerian path or circuit. We've seen that there are 30 edges and no circuit can be of length greater than 24. A slight modification of the argument which showed that also tells us that no path has length greater than 25. To get an example we therefore need to look at a different graph. The one in the accompanying figure will work.

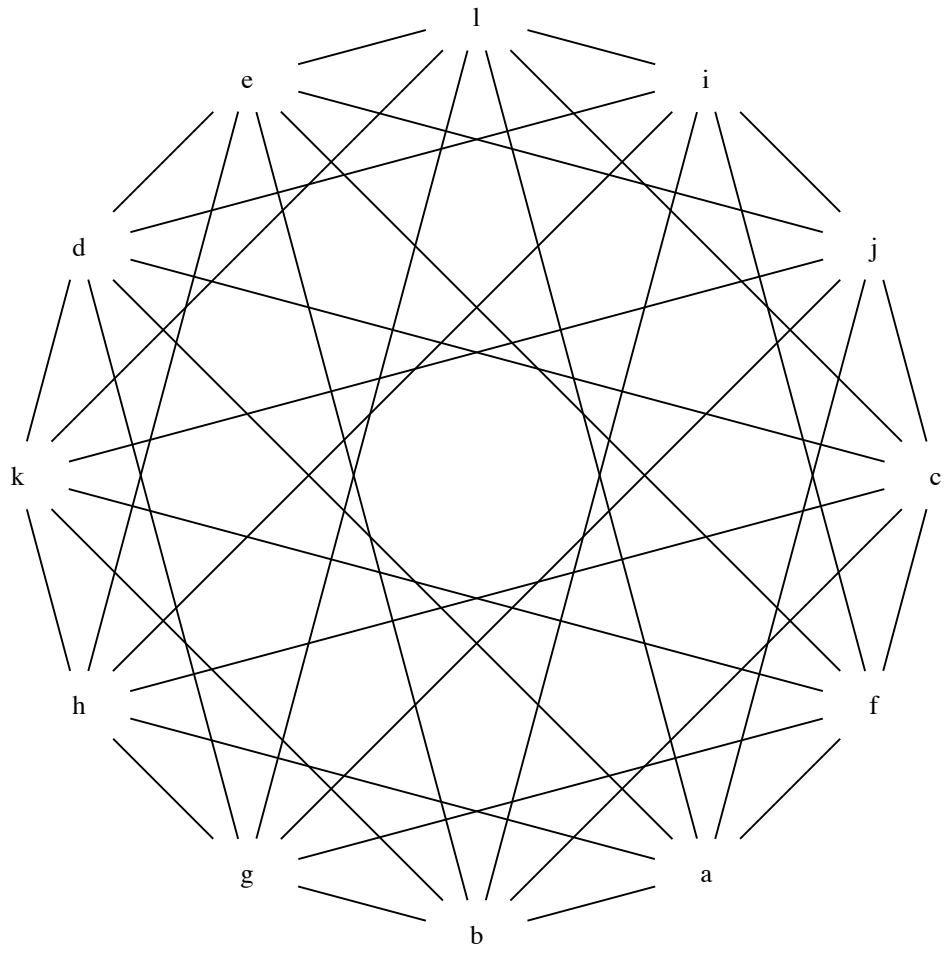


Figure 46: A bipartite regular graph

This is a regular bipartite graph with 12 vertices, each of degree 6. There are therefore 36 pairs of edges, so a circuit of length 36 must be Eulerian. One such example is the graph which visits the vertices a, b, e, l, a, d, k, l, c, j, k, b, i, j, a, h, i, l, g, h, k, f, g, j, e, f, i, d, e, h, c, d, g, b, c, f, and a that order, as shown in the following figure, where the edges are given the orientation in which they are traversed in the Eulerian trail.

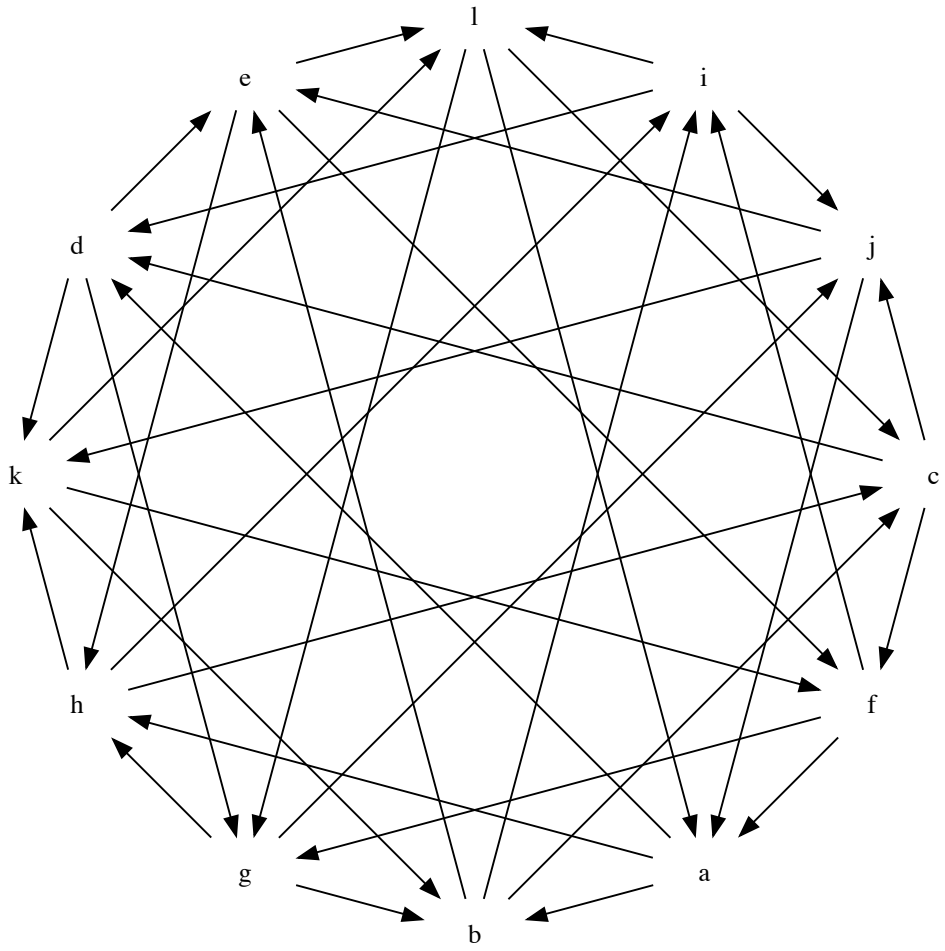


Figure 47: An Eulerian path in the bipartite regular graph

Connectedness

Given a graph with vertex set V we can define a relation S on V by saying that $(v, w) \in S$ if $v = w$ or there is a walk with initial vertex v and final vertex w. This is a reflexive relation, because we defined $(v, w) \in S$ to be true if $v = w$. It is also a transitive relation. In other words, if $(u, v) \in S$ and $(v, w) \in S$ then $(u, w) \in S$. If $u = v$ then (u, w) and (v, w) are the same, so it's clear that if $(v, w) \in S$ then $(u, w) \in S$. Similarly if $v = w$ then (u, v) and (u, w) are the same, so if $(u, v) \in S$ then $(u, w) \in S$. The only interesting case is therefore the one where there is a walk from u to v and a walk from v to w. In this case we can obtain a walk from u to w by concatenating the list of edges in the walk from u to v and the list of edges in the walk from v to w.

If the relation S is antisymmetric then we say the graph is a directed acyclic graph. In this case the set of vertices with the relation S form a partially ordered set.

A tree is just a directed acyclic graph in which there is at most one walk from any vertex to any other vertex. In the theory of undirected graphs we say that a graph is a tree if it's connected and has no simple circuit of length greater than two. These two seemingly different definitions are related as follows. An undirected graph is a tree, as defined for undirected graphs, if and only if its possible to choose a direction for each edge making it into a tree, as defined for directed graphs.

If the graph is undirected then S is symmetric, since we can then obtain a walk from v to w from a walk from w to v by reversing the order of the edges in the walk and reversing the direction of each edge. So in this case the relation S is an equivalence relation. The equivalence classes are called connected components. A non-empty undirected graph is called connected if it has only one connected component, i.e. if for every two distinct vertices there is a walk connecting them. All of the undirected graphs which have appeared so far are connected, except for the EU border graph, which has five connected components. One each with just Cyprus, Ireland and Malta as members, one with just Finland and Sweden, and one with all other EU states as members.

If two distinct vertices belong to the same equivalence class then there is a walk between them. Lengths of walks are natural numbers so there must

then be a shortest walk. If this walk had a vertex which appeared more than once then we could further shorten it by removing all the edges between its first appearance and its last, but then it wouldn't be a shortest walk, so there can be no repeated vertices. In other words the walk is a path. We already knew from the definition that any two vertices in a connected component are connected by a walk but the argument above shows that they are in fact connected by a path. This is a stronger statement since every path is a walk but not every walk is a path. It would have been a bad idea to define connected components in terms of paths though since this would have made it harder to prove the transitivity property.

Eulerian trails and circuits

Given a trail in an undirected graph we can form a subgraph by taking the same set of vertices in the original graph but keeping only those edges which appear in the trail. In the case of an Eulerian path we will then be keeping one edge from each pair. The diagram of this new, directed, graph will be the same as the diagram of the original graph, except each edge will have an arrow indicating its direction, as in our earlier bipartite regular graph example.

You may we recall that we've already seen a directed graph which selected one edge from each pair, as a way of showing that the graph is bipartite. That graph had the property that at each vertex the edges were either all outgoing or all incoming. In terms of degrees, for each vertex either the in-degree or out-degree is zero. This new directed graph is different. Here the in-degree and out-degree are always equal.

More generally, suppose we start from an Eulerian trail in an undirected graph and create a directed graph by keeping all the vertices and those edges belonging to the path, as above. Whenever a vertex appears in the interior of the trail, i.e. not as the initial or final vertex, it is the final endpoint of one edge and the initial endpoint of the following edge so the first of those edges contributes one to the in-degree and the latter contributes one to the out-degree. The initial edge contributes one to the out-degree of its initial endpoint and the final edge contributes one to the in-degree of its final endpoint. All of the contributions of any edge to the degrees of any vertex arise in one of the ways just described. So for all but the initial and fi-

nal vertices of the trail the in-degree and out-degree must be the same. For the initial vertex the in-degree is one less than the out-degree and for the final vertex it is one more, unless the initial and final vertex are the same, i.e. unless the trail is a circuit, in which case the in and out degrees at that vertex are again the same. Each edge in the directed graph corresponds to a pair of edges in the original undirected graph and each such edge contributes one to the degree of its endpoints so the degree of a vertex in the undirected graph is the sum of the in-degree and out-degree of that vertex in the directed graph. This degree is therefore even, except in the case of a trail which is not a circuit, in which case the degrees of the initial and final vertices are odd. There are therefore either zero or two vertices of odd degree in an undirected graph with an Eulerian trail. If there are zero then that trail, and all other Eulerian trails, are circuits. If there are two then that trail, and all other Eulerian trails, are not circuits. If the number of vertices of odd degree in an undirected graph is not equal to zero or two then there is no Eulerian trail.

In particular the regular graph we considered earlier with twelve vertices of degree five has no Eulerian trail since it has twelve odd vertices. We can also see that any Eulerian path on the graph we just considered is a circuit, since the number of odd vertices is zero.

Suppose we have an undirected graph all of whose vertices have even degree and at least one has positive degree. Then there is a trail of positive length through that vertex. The number of edges in trail is at most the number of total edges, which is finite, so there is a longest trail through that vertex. What can we say about this trail?

First of all, such a longest trail must in fact be a circuit. To see this we construct two subgraphs. Both have the same vertex set as the origin graph. The first has those edges which belong to the trail, along with the edges in the reverse direction. The second has all the other edges. These are both undirected graphs. The first graph was constructed to have an Eulerian trail. If the initial or final vertex of this trail had odd degree in the first subgraph then it would also have odd degree in the second subgraph, since the two degrees add up to the degree in the original graph. Zero is not an odd number so the degree in the second subgraph is positive, which means there is a pair of edges in the original graph with that vertex as their initial or final endpoint, neither of which belong to the trail. We could therefore

extend the trail by appending one or the other of these edges, either at the beginning, if the vertex is the initial vertex or the trail, or at the end, if it's the final vertex, to obtain a longer trail. Since our trail was chosen to be as long as possible this is impossible, so the degree of the initial and final vertices in the first subgraph is even and therefore those vertices are the same and the trail is a circuit.

Next, a longest trail contains one edge from each pair attached to any of its vertices. We use the same subgraphs as before. We've now established that the first subgraph has an Eulerian circuit and that the degrees of the vertices in an undirected graph with an Eulerian circuit are all even so the degrees of all vertices in the first subgraph are all even. We know that the degree of each vertex in the original graph is even and is the sum of its degrees in the two subgraphs so the degree in the second subgraph is also even. Suppose there were a vertex on the circuit which did not contain an edge from each pair connected to it. Then those edges would be in the second subgraph. The second subgraph thus has vertices of even order and this vertex has positive degree so by what we proved in the preceding paragraph, applied now to this subgraph, there is a circuit of positive length through this vertex. We could then splice this circuit in to the original trail to obtain a longer trail, but this is impossible, so the assumption that there is such a vertex is untenable.

So now we know that a longest trail through a vertex is necessarily a circuit and that it contains all edges connected to any of its vertices. Consider now a walk starting at the same vertex. The final vertex of this walk must be traversed by the circuit. This is proved by induction on the length of the walk. If the walk is of length 0 then the final vertex is the initial one and so is certainly traversed by the circuit. If the length is positive then we can assume, by induction, that circuit traverses the final vertex of the walk obtained by deleting the final edge of the original walk. But every edge through that vertex is then traversed by the circuit, including the edge we just deleted, so the final edge of the original path, and hence the final vertex, is traversed by the circuit.

Every vertex in the same component of the graph as the original vertex is connected to that vertex by a walk, and so is traversed by the circuit, as are all the edges connected to it. So a longest path through a vertex traverses every vertex and edge of that component. In particular, if the

graph is connected then the longest trail traverses every vertex and edge of the graph. It is therefore an Eulerian trail and, since we've already seen that it's a circuit, is an Eulerian circuit.

What we have just shown is that a connected undirected graph has an Eulerian circuit if and only if all of its vertices have even degree. There is a similar theorem for non-closed Eulerian trails. A connected undirected graph has such a trail if and only if exactly two of its vertices have odd degree. Those two vertices are the initial and final points of the trail. The trick to proving this is to consider the original graph as a subgraph of a larger graph, obtained by adding an extra vertex and pairs of edges from that vertex to the two odd vertices. The larger graph has vertices of even degree and so has an Eulerian circuit. This circuit goes through the added vertex. If we remove the edges in and out of this vertex then we obtain an Eulerian trail in the original graph.

Previously we saw that if there is an Eulerian circuit then the graph is connected and all vertices have even degree. We now have the converse, that if the graph is connected and all vertices have even degree then there is an Eulerian circuit. A similar statement applies to graphs with exactly two vertices of odd degree and Eulerian trails.

It's often said that proofs by contradiction are non-constructive but the one above does actually give an algorithm for finding Eulerian circuits:

- Choose a vertex.
- Starting at that vertex continue to an adjacent vertex, a vertex adjacent to that, etc., always avoiding edges which have already been used in either direction. Do this until there are no available edges left wherever you stopped.
- The vertex where you stopped must be the one where you started, so you have a circuit, but not necessarily an Eulerian one. If it's not Eulerian then there's a vertex somewhere along the path with edges you haven't used. Starting from that vertex continue to an adjacent vertex, a vertex adjacent to that, etc. When you can't continue any further you must have ended up at the vertex where you left the original circuit. Splice the new circuit into the old one at the point where it was first visited.

- Keep doing the preceding operation until there are no vertices on the circuit with available edges. At this point you have an Eulerian circuit.

Hamiltonian paths and circuits

A path is called Hamiltonian if it traverses every vertex exactly once. A circuit is called Hamiltonian if it traverses every vertex exactly once, except that the initial and final vertices are the same. The definition is similar to that of Eulerian trails and circuits, but the question of whether a graph has a Hamiltonian path or circuit turns out to be much more difficult to answer than the question of whether it has an Eulerian trail or circuit.

Some information is easy to obtain. K_n always has a Hamiltonian path and a Hamiltonian circuit. We can order the vertices however we like and then visit each one in order, since every pair of vertices is connected by an edge. To get a circuit we just append another edge from the last vertex in the path to the first.

For $K_{p,q}$ the answer depends on p and q. Any walk in $K_{p,q}$ alternately visits vertices from the set of p vertices and the set of q vertices, since there are no edges within either set. So at the end of any path in $K_{p,q}$ the number of edges visited from one set differs by at most one from the number visited from the other set. So there is no Hamiltonian path unless $|p - q| \leq 1$. Conversely, if this inequality is satisfied then we can find a Hamiltonian path. If $p = q$ then we can also find a Hamiltonian circuit.

The bipartite regular graph we used earlier as an example for Eulerian paths also has a Hamiltonian circuit, which is just a, b, c, d, e, f, g, h, i, j, k, l, a, which is shown in the accompanying diagram.

This Hamiltonian path is far from unique. We can obtain other ones just by visiting the vertices in clockwise or anticlockwise order.

The earliest example of a Hamiltonian path is the Knight's Tour problem in chess. The graph in question has the squares of the chessboard as vertices and vertices are adjacent if and only if they are a knight's move apart. A knight's tour is a set of moves visiting each square exactly once, i.e. a Hamiltonian path in the graph. The earliest known solutions are by al-Adli ar-Rumi and by Rudrata, and date to the ninth century. Again, these Hamil-

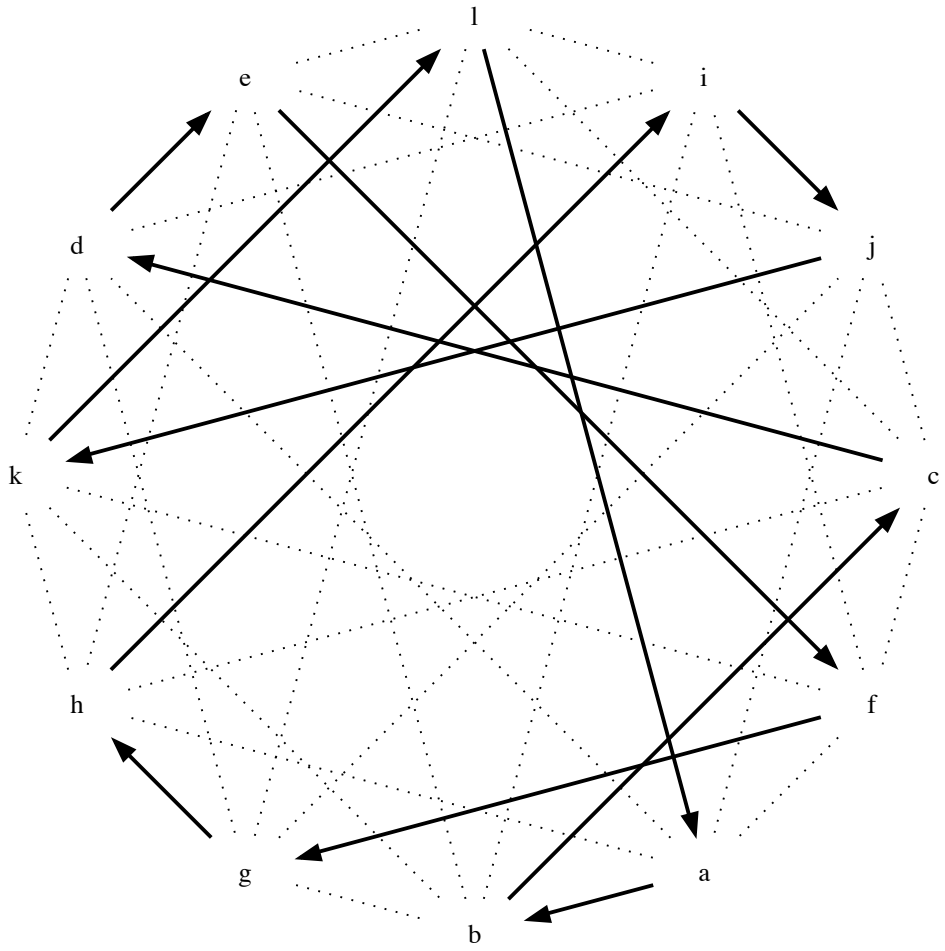


Figure 48: A bipartite regular graph

tonian paths are far from unique. In fact there are 19,591,828,170,979,904 of them.

Spanning trees

As we've already discussed, graphs do not necessarily have Hamiltonian paths. Connectedness is a necessary condition for the existence of a Hamiltonian path, but it's not a sufficient condition. Connected undirected graphs without self-loops do, however, always have spanning tree, i.e. a subgraph which is a tree and has every vertex of the original graph as a vertex.

The largest connected component of the EU border graph, for example, has the spanning tree shown in the accompanying diagram. Edges which belong to the spanning tree are shown in bold, while edges which belong to the original graph but not the spanning tree are dotted.

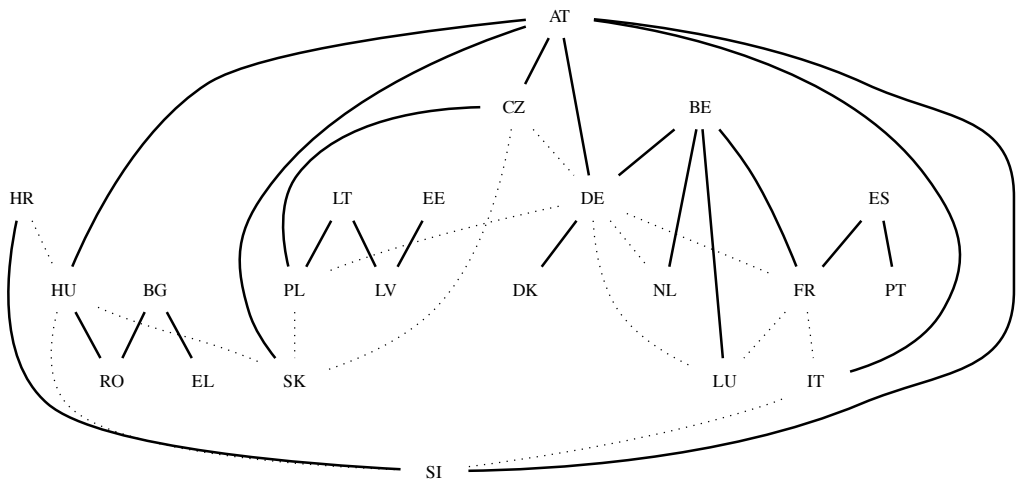


Figure 49: An undirected graph

To show that every connected graph has a spanning tree we use an argument similar to the one used earlier for the existence of Eulerian trails. For the rest of this paragraph whenever I refer to a tree I will mean a subgraph of the original graph which is a tree. The number of vertices in a tree is at most one less than the number of vertices in the graph, so there must be a

tree with a maximal number of vertices. If there were a vertex in the graph which was not in the tree then we could obtain a larger tree as follows. Pick a vertex in the tree and one not in the tree. The graph is connected so there is a walk from the first vertex to the second. Consider the last vertex of the tree which belongs to this walk. The next edge connects that vertex to a vertex not in the tree. Adjoining that edge to the tree gives a graph which is still connected and has no loops and so is a tree. It would therefore be a larger tree, but we chose our tree to be maximal, so this can't happen. In other words, there is no vertex in the tree but not in the graph, so the tree is a spanning tree.

There are, in general, many possible spanning trees. It's common in applications that there is a cost function on the edges of the graph and that one wants to minimise the cost of the tree, i.e. the sum of the costs of the edges in the tree, among all the spanning trees of the graph. Such a cost-minimising spanning tree is called a minimal spanning tree. The existence of a minimal spanning tree is easy to prove. Spanning trees are determined by their edges, which are a subset of the edges of the original graph. There are only finitely many edges in the original graph and so only finitely many spanning trees. The total cost determines an ordering of spanning trees and we've already seen that orderings of finite sets have minimal elements. If you actually want to find a minimal spanning tree then finding all spanning trees, computing their total costs, and then choosing one with the lowest cost is not an efficient algorithm. A number of efficient algorithms are known though.

Abstract algebra

Binary operations

If A is a set then a function from A^2 to A is called a binary operation. Examples include

- $\wedge$ on Boolean truth values
- $\vee$ on Boolean truth values
- $\supset$ on Boolean truth values

- $+$ on the natural numbers
- $\cdot$ on the natural numbers
- the maximum operation on natural numbers
- the minimum operation on natural numbers
- $\cap$ on the power set of a given set
- $\cup$ on the power set of a given set
- $\setminus$ on the power set of a given set
- $\circ$ on the set of functions from a given set to itself
- $\circ$ on the set of relations on a given set to itself
- $\circ$ on the set of homomorphisms from a graph to itself
- $\circ$ on the set of isomorphisms from a graph to itself
- the concatenation operation on lists all of whose items belong to a given set

Functions are left total, so we can't define subtraction or division as binary operations on the natural numbers. We can define subtraction as a binary operation on a larger set, like the set of integers, rationals or reals. We can't define division as a binary operation on any of these sets, at least if we want the relation $(x/y) \cdot y = x$ to hold for all x and y, because of the problems with division by zero.

The notation used for binary operations varies. Most of the operations above are usually written with an infix notation, like $x \cdot y$, $A \cup B$, or $f \circ g$. Maximum and minimum are usually written with functional notation, like $\max(x, y)$. Arguably this should be $\max((x, y))$ with one set of parentheses identifying function arguments and the other identifying an ordered pair, since this is a function on ordered pairs, but in reality no one uses that notation. The infix notation $x \wedge y$ for maximum and $x \vee y$ for minimum is sometimes used. This is consistent with the notation for Boolean operators provided you accept that falsehood is greater than truth. Notation for concatenation is not completely standardised. Functional notation is used by some authors. Others use an infix notation, often with no actual symbol in between, like vw for the list consisting of the items of v followed by those

of w. I'll use a mix of notations, but will mostly prefer functional notation when described properties of general binary operations and whatever notation is most commonly used for specific binary operators when they appear as examples.

A binary operation f on a set A is called associative if

$$f(x,f(y,z)) = f(f(x,y),z)$$

for all x, y and z in A. It is called commutative if

$$f(x,y) = f(y,x)$$

for all x and y in A.

We can apply the associativity property multiple times to show that, for example

$$f(w,f(x,f(y,z))) = f(f(w,x),f(y,z)) = f(f(f(w,x),y),z).$$

This is usually easier to follow with an infix notation. For example, the previous calculation applied to the union operator for sets is

$$A \cup (B \cup (C \cup D)) = (A \cup B) \cup (C \cup D) = ((A \cup B) \cup C) \cup D.$$

The parentheses tell you in what order the union operator is to be applied but the equation essentially tells you that the order doesn't matter. Or at least it tells you that the order doesn't affect the final result. In a computational problem the order may have a very noticeable affect on the time or resources required. Matrix multiplication, for example, is associative, in the sense that if L, M and N are matrices such that the M has as many rows as L has columns and as many columns as N has rows then

$$(LM)N = L(MN).$$

Without those conditions on the numbers of rows and columns the products are not defined. Suppose L is has m rows and n columns while N has p rows and q columns. The number of multiplications needed to compute LM is mnp and the number of further multiplications needed to compute $(LM)N$ is mpq, so the left hand side requires $mp(n+q)$ multiplications. Similarly, number of multiplications needed to compute MN is npq and the

number of further multiplications needed to compute $L(MN)$ is mnq so the total number for the right hand side is $(m+p)nq$. These numbers might be very different. In the case of a square matrix followed by a column vector, thought of as a matrix with a single column, and then a row vector, thought of as a matrix with a single row, we would have $m = n = q$ and $p = 1$ so the left hand side needs $2m^2$ operations while the right hand side needs $m^3 + m^2$ operations. Quite a bit of computational linear algebra is devoted to figuring out the most efficient ways to apply associativity.

Of the examples above, $\supset$ and $\setminus$ are neither associative nor commutative. $\circ$ and concatenation are associative but not commutative. The remaining ones are all associative and commutative. It's certainly possible to construct examples of operations which are commutative but not associative, but naturally occurring examples are somewhat rare. One is the nand operator, $\overline{\wedge}$, from Boolean algebra, which we briefly considered in the context of the Nicod formal system.

Semigroups

A pair (A,f), where A is a set and f is an associative binary operation on A, is called a semigroup.

If (A,f) is a semigroup and B is a subset of A then we can restrict f to get a function from B^2 to A. If the range of this function is a subset of B, i.e. if $f(x,y) \in B$ whenever $x \in B$ and $y \in B$, then this restriction is a binary operation on B. It is necessarily associative because if $x \in B$, $y \in B$, and $z \in B$, then $x \in A$, $y \in A$, and $z \in A$, and so

$$f(x,f(y,z)) = f(f(x,y),z),$$

by the associativity of f on A. If $f(x,y) \in B$ whenever $x \in B$ and $y \in B$ then we say that B is a subsemigroup of A. As an example, the set of even natural numbers, with addition as the operation, is a subsemigroup of the natural numbers, also with addition. Not every subset is a subsemigroup though. The set of prime numbers is not a subsemigroup because the sum of prime numbers needn't be prime.

There is a general associativity property for semigroups which I hinted at with the equation

$$f(w,f(x,f(y,z))) = f(f(w,x),f(y,z)) = f(f(f(w,x),y),z)$$

above. In stating this it's convenient to use the word "multiplication" for the function f, even though multiplication is just one of the possible binary operations we might consider, and to refer to the result of applying f to two members of A as the "product" of those elements. With this convention the generalised associativity property we want to prove says that the order in which multiple multiplications is performed doesn't change the final product.

One way to state it more precisely, related to our discussion of parsing earlier, is in terms of binary trees. Given a list of n members of A and a binary tree with n leaves we can compute a corresponding product by filling in the list items in the leaves and then proceeding up the tree to the root, multiplying the values of a node's children to obtain its value. There are, for example, five possible shapes for a binary tree with four leaves, which you found in Assignment 0. Each of these gives a different way to compute the product of a list (w, x, y, z) of members of A, illustrated in the five accompanying figures.

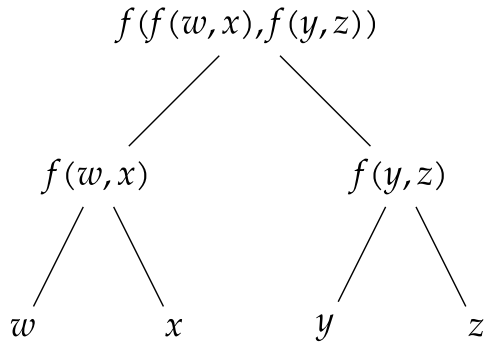


Figure 50: A tree for $f(f(w,x),f(y,z))$

One way to see that these are all give the same result for any associative operation is to look at the accompanying graph, where the five vertices are the five products and the edges connect those products which can be shown equal with a single application of the associative law.

The fact that this graph is connected implies that we can show any two are equal via repeated applications of the associative law, since any walk tells us the order in which we need to apply the associative law in order to

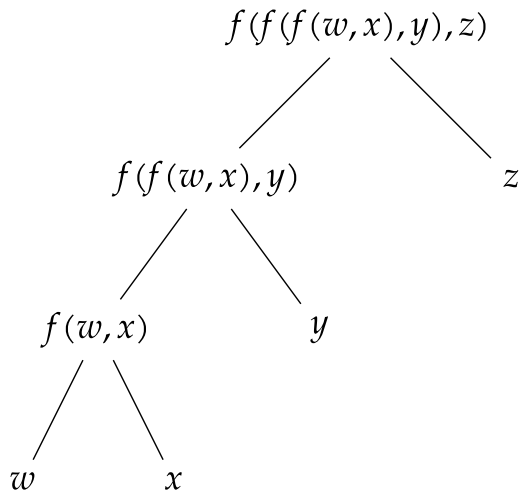


Figure 51: A tree for $f(f(f(w,x),y),z)$

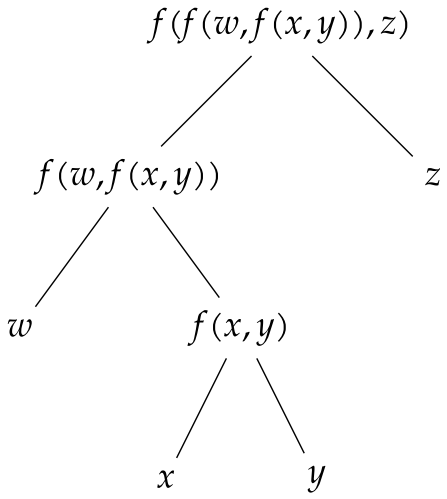


Figure 52: A tree for $f(f(w,f(x,y)),z)$

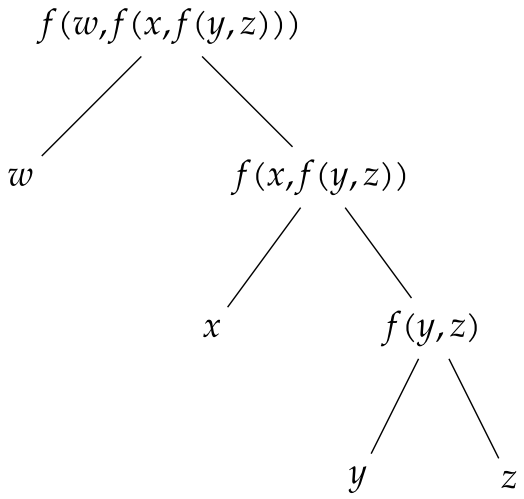


Figure 53: A tree for $f(w,f(x,f(y,z)))$

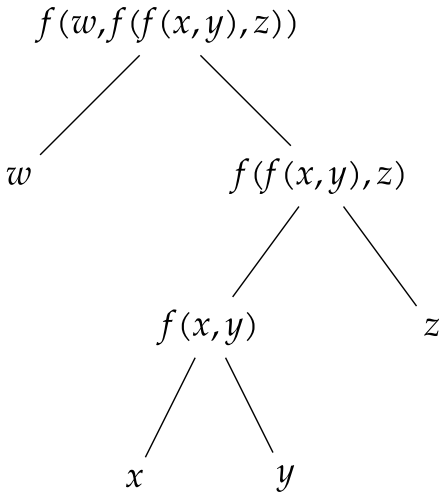


Figure 54: A tree for $f(w,f(f(x,y),z))$

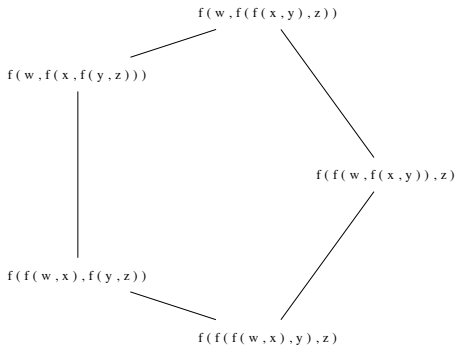


Figure 55: A graph for products of length 4

get from the product labeling its initial vertex to the one labeling its final vertex.

Nothing but boredom prevents us from doing the same thing for products of size n for any larger value of n. The accompanying diagram shows the graph for $n = 5$.

It's harder to see, visually, that this graph is connected, but it is. We'd like an argument which applies to all values of n though, rather than treating each one separately.

The easiest way to prove that all possible products for a given list are equal is to prove that each is equal to some particular product. We'll call the product where we take our list and continue multiplying the two leftmost items until there's only a single item the leftmost product. In our earlier example, starting with the original list (w, x, y, z) we would go through the steps $(f(w, x), y, z)$, then $(f(f(w, x), y), z)$, then $(f(f(f(w, x), y), z))$ so the leftmost product would be $f(f(f(w, x), y), z)$, the second of the trees shown previously, and the one which leans to the left the most. This is the product which we will show that all others are equal to. We won't define the leftmost product of the empty list but the leftmost product of a list with only a single item is just that item, which is the trivial case of the procedure described above of multiplying the leftmost items until only a single item remains.

We can start with the special case that the product of two leftmost products is a leftmost product. In other words if p is the leftmost product of some list

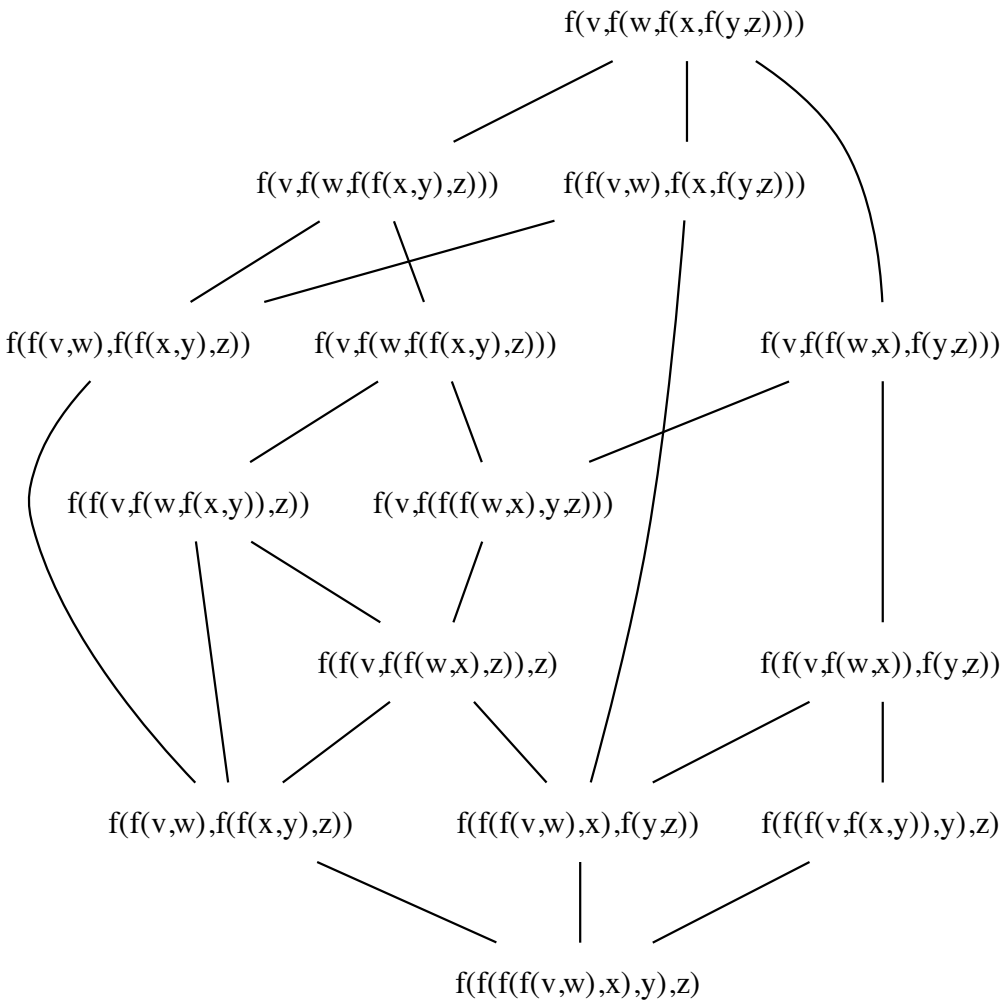


Figure 56: A graph for products of length 5

P of members of A and q is the leftmost product of some list Q of members of A then $f(p,q)$ is equal to the leftmost product of the concatenation of the list P and the list Q. If this were not true then there would be a shortest list Q for which it failed. We're not defining lists of length zero so Q is either of length one or length greater than one. If it's of length one then it is just (q) and the concatenation of P and Q is the list P with a q appended at the end. When we compute its leftmost product as described above we first multiply all the items to the left of this final q, obtaining p and then multiply it with q, obtaining $f(p,q)$, which is what we're meant to find, so the property cannot fail when Q is of length one. It follows that Q must of length greater than one. Let R be the list consisting of all but the last item in Q and let s be the last item. Let r be the leftmost product of R. Then

$$q = f(r,s)$$

so

$$f(p,q) = f(p,f(r,s))$$

and hence, by the associativity property,

$$f(p,q) = f(f(p,r),s).$$

Now R and (s) are shorter then Q and Q is a list of the least length for which the product of the leftmost products is not the leftmost product so $f(p,r)$ is the leftmost product of the concatenation of P and R. and $f(f(p,r),s)$ is the leftmost product of the concatenation of that list with (s), which is the concatenation of P with Q. In other words $f(p,q)$ is the leftmost product of the concatenation of P with Q. We've just seen that even if we assume we have a counterexample to the statement that the product of the leftmost products is the leftmost product of the concatenation then we find out that it isn't one. There therefore isn't a counterexample. This concludes the proof of the special case.

We can now continue to the proof of the general case. Suppose there is a product of a list which is not equal to the leftmost product. There must then be a shortest such list. We haven't defined products for lists of length zero and there's only one possible product for a list of length one so our list must be of length greater than one and so must be of the form $f(p,q)$ where P and Q are lists whose concatenation is this list and p and q are some products of P and Q. But P and Q are shorter than the whole list and

so any product for them is equal to the leftmost product. The previous paragraph therefore shows that $f(p,q)$ is equal to the leftmost product of the concatenation of P and Q, which is the list we started with. So again, even if we assume the existence of a counterexample we find that it isn't one.

So now any two products for a list are equal to the leftmost product and therefore equal to each other, and therefore all products for a list are equal. We can therefore forget about the order of products in a semigroup.

Identity elements, monoids

Suppose (A,f) is a semigroup, i.e. that f is an associative binary operation on the set A. A member i of A is said to be an identity element if for all $x \in A$ we have

$$f(i,x) = x$$

and

$$f(x,i) = x.$$

Going back to our earlier examples of associative operations we can see that all but one of them have identity elements.

- $\wedge$ on Boolean truth values: the value false is an identity element.
- $\vee$ on Boolean truth values: the value true is an identity element.
- $+$ on the natural numbers: 0 is an identity element.
- $\cdot$ on the natural numbers: 1 is an identity element.
- the maximum operation on natural numbers: 0 is an identity element.
- the minimum operation on natural numbers: there is no identity element.
- $\cap$ on the power set of a given set: the whole set is an identity element.
- $\cup$ on the power set of a given set: the empty set is an identity element.
- $\circ$ on the set of functions from a given set to itself: the identity function is an identity element.

- $\circ$ on the set of relations on a given set: the identity function is an identity element.
- the concatenation operation on lists all of whose items belong to a given set: the empty list is an identity element.

For the ones which do have an identity element it's straightforward in each case to see that the given element is indeed an identity. For the minimum the non-existence of an identity is the statement that there is no natural number i such that

$$\min(i, x) = x$$

and

$$\min(x, i) = x$$

for all x, which is clear because the equations above would fail for $x = i + 1$.

I've referred to an identity rather than the identity above to allow for the possibility that there might be more than one, but in fact this can't happen. Suppose i and j are identity elements. Then

$$f(i, j) = j$$

because i is an identity and

$$f(i, j) = i$$

because j is an identity so

$$i = j.$$

A semigroup with an identity element is called a monoid, so all the semigroups listed above, except for the minimum operation on the natural numbers, are monoids.

A subsemigroup of a monoid which contains the identity element is also a monoid, with the operation being the restriction of the original operation and the identity being the identity from the larger monoid. The subsemigroup is then called a submonoid. The set of even natural numbers, considered earlier as a subsemigroup of the natural numbers, is a submonoid. Not all subsemigroups of a monoid are submonoids though.

Another example of a monoid is what's called the bicyclic semigroup. The set in this case is the set N^2 of ordered pairs of natural numbers and the

binary operation is

$$f((a,b),(c,d)) = (a + c - \min(b,c), b + d - \min(b,c)).$$

I will skip the straightforward but tedious verification that this operation is associative and therefore that this is a semigroup. It is not commutative since

$$f((0,1),(1,0)) = (0,0)$$

while

$$f((1,0),(0,1)) = (1,1).$$

We have

$$f((0,0),(x,y)) = (x,y)$$

and

$$f((x,y),(0,0)) = (x,y)$$

for all (x,y) so $(0,0)$ is an identity element. This is therefore not just a semigroup but a monoid. It would make more sense therefore to refer to it as the bicyclic monoid, but for historical reasons it is referred to as the bicyclic semigroup. That name isn't wrong, since it is a semigroup, but it's imprecise.

Inverse elements and groups

Suppose (A,f) is a monoid with identity element i. $y \in A$ is said to be an inverse to $x \in A$ if

$$f(x,y) = i$$

and

$$f(y,x) = i$$

and x is then said to be invertible. It's immediate from the definition that y is an an inverse to x if and only if x is an inverse to y. Also, the identity element is its own inverse.

As previously with identity elements I've deliberately written "an" rather than "the" to allow for the possibility that there might be more than one but we can show that this can't happen. Suppose y and z are inverses to x. Then we have the following equations:

$$f(y,i) = y,$$

$$f(x,z) = i,$$
$$f(y,f(x,z)) = y,$$
$$f(y,f(x,z)) = f(f(y,x),z),$$
$$y = f(f(y,x),z),$$
$$f(y,x) = i,$$
$$y = f(i,z),$$
$$f(i,z) = z,$$
$$y = z.$$

Each equation in this chain is one of the following: substitution one of two equal values for the other, the definition of an identity element applied to i, the definition of an inverse applied to y or to z, or the associativity of f.

We can go through our list of monoids and identify the invertible elements, if any, and their inverses. In almost all cases the only invertible element is the identity. The only exception is $\circ$ on the set of functions from a given set to itself, or on the set of relations on a set, where the identity function is the identity element and the bijective functions are the invertible elements. The inverse of a function is the inverse function.

In the case of the addition operation on the natural numbers we can extend the operation to a larger monoid in such a way that every element acquires an inverse. The larger monoid in this case is the set of integers and the inverse of x is just $-x$. In the other examples this is not possible, though this is easier to see in some cases than in others.

If a and b are invertible elements of a monoid (A,f) $f(a,b)$ is also invertible. More precisely, let c be the inverse of a and d the inverse of b. Then $f(d,c)$ is an inverse of $f(a,b)$. The proof is as follows.

$$f(f(a,b),f(d,c)) = f(a,f(b,f(d,c)),$$
$$f(a,f(b,f(d,c)) = f(a,f(f(b,d),c)),$$
$$f(a,f(f(b,d),c)) = f(a,f(i,c)),$$
$$f(a,f(i,c)) = f(a,c),$$

and

$$f(a,c) = i,$$

so

$$f(f(a,b),f(d,c)) = i.$$

The same argument, with a and d swapped and b and c swapped gives

$$f(f(d,c),f(a,b)) = i.$$

A monoid where every element is invertible is called a group.

Given a monoid the set of invertible elements has, as we just saw, the property that $f(a,b)$ is a member if a and b are, and so is a subsemigroup. It has the identity as a member and so is a submonoid, and in particular is a monoid. Every element is invertible so it's actually a group.

A subset of a group is called a subgroup if it is a submonoid and has the property that if x is a member then so is the inverse of x. A subgroup is, as the name suggests, a group.

In all but one of the examples of monoids considered above the set of invertible elements is a trivial group, i.e. a group with no elements other than the identity. The exception is the monoid of functions from a set to itself, where we get the group of bijective functions on that set. In the important special case where the set is finite the group is called a permutation group. If the set on which our functions are defined had n elements then the corresponding permutation group has $n!$ elements.

Other important examples of groups are the integers, with the operation of addition, or the non-zero rationals, with the operation of multiplication, or the set of invertible matrices with a given number of rows and columns, with the operation of matrix multiplication. Another example of a group is the set of rigid motions of Euclidean space, i.e. the set of rotations, reflections, translations and the identity.

One common source of groups is symmetries of some structure. For example the set of isomorphisms of a graph is a group, with composition as the operation and the identity function as the identity. The permutation groups arise in this way, as the isomorphism groups of the complete graphs.

Homomorphisms

Suppose (A, f) and (B, g) are semigroups. A function h from A to B is called a semigroup homomorphism if it has the property that

$$g(h(x), h(y)) = h(f(x, y))$$

for all $x \in A$ and $y \in A$.

A semigroup homomorphism need not be a bijective function but if it is then its inverse function is also a semigroup homomorphism. In this case it's called a semigroup isomorphism and the two semigroups are called isomorphic. In the particular case where both semigroups are the same the isomorphisms are called automorphisms.

If B is a subsemigroup of A and g is the restriction of f to B then the inclusion function is a semigroup homomorphism.

For a less trivial example, consider the natural numbers N, with maximum as the operation, and the bicyclic semigroup N^2 considered earlier. Then the function h defined by

$$h(x) = (x, x)$$

is a semigroup homomorphism, since you can easily check that if g is the operation defined earlier,

$$g((a, b), (c, d)) = (a + c - \min(b, c), b + d - \min(b, c)),$$

then

$$g((x, x), (y, y)) = (\max(x, y), \max(x, y)).$$

Suppose (A, f) and (B, g) are monoids. A function h from A to B is called a monoid homomorphism if it is a semigroup homomorphism and $h(i) = j$, where i is the identity element of (A, f) and j is the identity element of (B, g).

A monoid homomorphism need not be a bijective function but if it is then its inverse function is also a monoid homomorphism. In this case it's called a monoid isomorphism and the two monoids are called isomorphic. In the particular case where both monoids are the same the isomorphisms are called automorphisms.

The inclusion of submonoid in a monoid is a monoid homomorphism. The semigroup homomorphism from the natural numbers to the bicyclic monoid considered above is a monoid homomorphism since we've already seen that it's a semigroup homeomorphism and we have $h(0) = (0, 0)$.

Another example of a monoid homomorphism is the length function on lists of items in a given set. This is a homomorphism from the set of lists, with the operation of concatenation, to the set of natural numbers, with the addition operation.

Suppose (A, f) and (B, g) are groups. A function h from A to B is called a group homomorphism if it is a monoid homomorphism. One could add the condition that h takes inverses to inverses but that's redundant.

A group homomorphism need not be a bijective function but if it is then its inverse function is also a group homomorphism. In this case it's called a group isomorphism and the two groups are called isomorphic. In the particular case where both groups are the same the isomorphisms are called automorphisms.

Note that the sets of semigroup automorphisms of a semigroup, monoid automorphisms of a monoid and group automorphisms of a group are all groups.

Quotients

Suppose (A, f) is a semigroup and R is an equivalence relation on A with the property that if

$$(u, x) \in R$$

and

$$(v, y) \in R$$

then

$$(f(u, v), f(x, y)) \in R.$$

Let B be the set of equivalence classes for the relation R. Let H be the set of $(x, C) \in A \times C$ such that $x \in C$. Each member of A is a member of an equivalence class so H is left total. Every member of A is a member of only one equivalence class so H is right unique. In other words H is a function. This means it's safe to use standard functional notation so I'll write

$C = h(x)$ in place of $(x, C) \in H$ or $x \in C$ from now on. Let G be the relation from B^2 to B consisting of those $((C, D), E) \in B^2 \times E$ for which there are $x \in C$, $y \in D$ and $z \in E$ with $z = f(x, y)$. For any $(C, D) \in B^2$ we can find $x \in C$ and $y \in D$ since R is an equivalence relation. Setting $z = f(x, y)$ and $E = h(z)$ we have $((C, D), E) \in G$, so G is left total. Suppose $((C, D), E) \in G$ and $((C, D), F) \in G$. The fact that $((C, D), E) \in G$ means there are $u \in C$, $v \in D$ and $w \in E$ such that $w = f(u, v)$ and the fact that $((C, D), F) \in G$ means there are $x \in C$, $y \in D$ and $z \in F$ such that $z = f(x, y)$. Since $u \in C$ and $x \in C$ we have $(u, x) \in R$ by the definition of an equivalence class. Similarly, $(v, y) \in R$. Because our assumptions about f and R we then have $(f(u, v), f(x, y)) \in R$, i.e. $(w, z) \in R$. Since $w \in E$ and $z \in F$ it then follows from the definition of an equivalence class that $E = F$. So if $((C, D), E) \in G$ and $((C, D), F) \in G$ then $E = F$. In other words G is right unique. We've already seen that it's left total so its a function. As with H I'll now switch to functional notation and write $E = h(C, D)$ in place of $((C, D), E) \in G$ from now on. For any members x and y of A if we set $z = f(x, y)$ then $((h(x), h(y)), h(z)) \in G$, or, in functional notation $h(z) = g(h(x), h(y))$. We can write this as

$$h(f(x, y)) = g(h(x), h(y)).$$

This equation is the one which appeared in the definition of a semigroup homomorphism.

Functions from B^2 to B are binary operations on B so G is a binary operation. I claim that it's associative. Suppose C, D and E are members of B. Equivalence classes are always non-empty so there are members x, y and z of A such that $x \in C$, $y \in D$ and $z \in E$. By the associativity of f we have

$$f(f(x, y), z) = f(x, f(y, z)),$$

from which it follows that

$$h(f(f(x, y), z)) = h(f(x, f(y, z))).$$

Now

$$h(f(f(x, y), z)) = g(h(f(x, y)), g(z))$$

and

$$h(f(x, y)) = g(h(x), h(y))$$

so

$$h(f(f(x, y), z)) = g(g(h(x), h(y)), h(z)).$$

Also, $h(x) = C$, $h(y) = D$ and $h(z) = E$, so

$$h(f(f(x,y),z)) = g(g(C,D),E).$$

A very similar argument shows that

$$h(f(f(x,y),z)) = g(C,g(D,E)).$$

We therefore have

$$g(g(C,D),E) = g(C,g(D,E)).$$

In other words, g is an associative operation on B and (B,g) is a semigroup.

The semigroup (B,g) is called the quotient of the semigroup (A,f) by the equivalence relation R. We've already seen that

$$h(f(x,y)) = g(h(x),h(y))$$

so h is a semigroup homomorphism.

It is straightforward to check that if i is an identity for (A,f) then $j = h(i)$ is an identity for (B,g). To see this, suppose $C \in B$. Equivalence classes are non-empty subsets so there is an $x \in C$, i.e. an $x \in A$ such that $x \in C$. Then

$$g(C,j) = g(h(x),h(i)),$$

$$g(h(x),h(i)) = h(f(x,i)),$$

$$f(x,i) = x$$

and

$$h(x) = C$$

so

$$g(C,j) = C.$$

A similar argument shows that $g(j,C) = C$, so j is an identity for (B,g). So if (A,f) is a monoid then (B,g) is also a monoid and h is a monoid homomorphism.

Suppose (A,f) is a group. If $C \in B$ then there is an $x \in C$, i.e. an x such that $h(x) = C$. Every element of a group is invertible so there is a $y \in A$ which is an inverse of x. Then

$$g(C,h(y)) = g(h(x),h(y)),$$

$$g(h(x), h(y)) = h(f(x, y))),$$

$$f(x, y) = i,$$

and

$$h(i) = j$$

so

$$g(C, h(y)) = j$$

and similarly $g(h(y), C) = j$ so $h(y)$ is an inverse of C, which is therefore invertible. C was an arbitrary element of B so every element of B is is invertible. In other words, (B, g) is a group.

An argument similar to the two above shows that if f is a commutative binary operation on A then g is a commutative binary operation on B, so if (A, f) is a commutative semigroup, monoid or group then (B, g) is a commutative semigroup, monoid or group.

Integers and rationals

I've referred to the integers informal and rationals a few times but haven't defined them. The usual construction is via equivalence classes, as above.

Consider the operation

$$f((a, b), (c, d)) = (a + c, b + d)$$

on N^2. Note that this is a different operation than the one which I used in defining the bicyclic semigroup.

We can define an equivalence relation R on N^2 by $((a, b), (c, d)) \in R$ if and only if $a + d = b + c$. It is straightforward to show that this equivalence relation is compatible in the sense we considered in the previous section so the equivalence classes form a commutative monoid. This monoid turns out to be a group, even though N^2 itself is not a group. The inverse element to (a, b) is (b, a). This is easily verified because

$$f((a, b), (b, a)) = (a + b, b + a)$$

and

$$((a + b, b + a), (0, 0)) \in R.$$

The group of these equivalence classes is called the integers. The function g is just addition. The function h is just $h(a, b) = a - b$. Of course to make this look like the integers we need not just addition but also subtraction and multiplication, and also our $\leq$ relation. We don't need to, and indeed can't, define $=$ because it's already defined. Equivalence classes are sets and equality of sets is determined by the Axiom of Extensionality. Subtraction is defined by adding the inverse.

Multiplication is more complicated. One would like to define it via the equation

$$(a - b) \cdot (c - d) = [(a \cdot c) + (b \cdot d)] - [(a \cdot d) + (b \cdot c)]$$

which in terms of h is

$$h(a, b) \cdot h(c, d) = h((a \cdot c) + (b \cdot d), (a \cdot d) + (b \cdot c)).$$

This isn't quite suitable as a definition though. What we really need to do is to define $x \cdot y$ where x and y are equivalence classes. There certainly are a and b such that $h(a, b) = x$ but there are many such pairs (a, b). For example $h(a + 1, b + 1) = x$. We need a definition in terms of x itself, not a particular element of the equivalence class. I won't give the details, but the idea is similar to the way we defined the g in terms of G in the previous section. One first defines a binary relation and then shows that it is left total and right unique and so defines a function.

The procedure for the $\leq$ relation is similar. We would like to define it by saying that

$$a - b \leq c - d$$

i.e.

$$h(a, b) \leq h(c, d)$$

if and only if

$$a + d \leq b + c$$

but this doesn't work because a and b are not uniquely determined by $h(a, b)$ and c and d are not uniquely determined by $h(c, d)$. We can resolve this in a similar way to the one used for multiplication though.

There is one thing which is quite strange about this implementation of the integers though. The natural numbers should be a subset of the integers

but they aren't. In this implementation integers are sets of ordered pairs of natural numbers. The best we can do is to define a monoid homomorphism k from the natural numbers to the integers, by

$$k(n) = h((n, 0)).$$

This is not just a monoid homomorphism but can be shown to preserve multiplication and the $\leq$ relation as well, so in some sense the range of k serves as an alternative implementation of the integers.

The usual way to deal with this problem is to ignore it. If that makes you uncomfortable then there are two more honest, but more complicated approaches. The first is declare that the range of k is the set of natural numbers and that the things we previously called natural numbers are a distinct set, though one with the same behaviour as the true natural numbers, and that the integers are defined in terms of this other set, and the true natural numbers are a subset of the integers. This shouldn't be particularly alarming. We already saw multiple implementations of the natural numbers and this is just another one. An alternative approach is to keep the natural numbers unchanged but to define the integers differently, specifically as the union of the natural numbers and the complement in the set of equivalence classes above of the range of k. Operations on this new implementation of the integers have a more complicated definition. Essentially we take the operands, apply k to any which are natural numbers, apply the operation on equivalence classes, check whether the result is in the range of k, and replace it with n if it's $k(n)$. This is somewhat awkward but it works, in the sense that it gives a set and operations which behave correctly and have the natural numbers, as defined previously, as a subset.

Just as we can construct the integers from the natural numbers by a quotient construction we can construct the rationals from the integers. If Z is the group of integers then we define a binary operation on $Z \times (Z \setminus \{0\})$ by

$$f((a, b), (c, d)) = ((a \cdot d) + (b \cdot c), c \cdot d)$$

and an equivalence relation by

$$((a, b), (c, d)) \in R$$

if and only if

$$a \cdot d = b \cdot c.$$

These definitions are designed to make the homomorphism h satisfy

$$h(a,b) = a/b,$$

once we have defined division.

I will skip all the details of this construction. It does have an analogous problem to the one we encountered earlier in this section though. The integers are not a subset of the rationals. Again we can either choose to ignore the problem, or resolve it one of the ways discussed earlier, by redefining the integers or the rationals. As before this involves a homomorphism, this time from the integers to the rationals, which gives us an isomorphic copy of the integers in the rationals. This time the homomorphism is $k(x) = h(x,1)$.

More complicated number systems are developed in a similar way. The construction of the real numbers from the rationals is particularly difficult but there are a number of standard methods and most of them use the quotient construction with some appropriate choice of group and equivalence relation. Once you have the reals you can easily construct the complex numbers, again by the quotient construction.

The power function

Suppose (A,f) is a semigroup and $x \in A$. Then there is a function from the positive natural numbers to A obtained by taking the product of n copies of x, with the convention discussed in an earlier section of using the word "product" to denote the result of repeated applications of the binary operation f. We don't need to specify the order because of the generalised associativity property proved in that section. Addition is an associative operation on the positive natural numbers, making them into a semigroup, and this function is a semigroup homomorphism. The proof of this depends on the generalised associativity property. This function is not generally written with functional notation but with exponential notation. The value of the function corresponding to a particular x at a positive natural number n is written x^n. The property that the function is a semigroup homomorphism is, in this notation,

$$f(x^m, x^n) = x^{m+n}.$$

This notation is confusing in some examples. If, for example, the semigroup in question is the natural numbers with the operation of addition then x^n is not in fact the number normally denoted by that expression but rather is $n \cdot x$. If the operation is the maximum then x^n is just x. With sets and the operation of union or intersection A^n would just be A, rather than the set of lists of length n of items in A. Unfortunately the exponential notation is too well established to abolish entirely, but I'd suggest not using it where it conflicts with an established notation.

In a commutative semigroup it's possible to prove, by induction on n, that $f(x^n, y^n) = f(x, y)^n$. This is not generally true in a noncommutative semigroup though.

If our semigroup is a monoid then we can extend the function described above from the positive natural numbers to all natural numbers by defining x^0 to be the identity. The resulting extension is a monoid homomorphism. If the monoid is a group then we can extend it still further, by defining x^{-n} to by y^n where y is the inverse of x. This extended function is a group homomorphism. In this case we have the useful relation

$$f(x, y)^{-1} = f(y^{-1}, x^{-1}).$$

Note the reversal of the order of the arguments. This identity was in fact proved earlier, but in a different notation, in the course of proving that the product of invertible elements is invertible.

Notation

If you only consider one semigroup, monoid or group, or if you consider only a particular one and its subsemigroups, submonoids or subgroups, then it's convenient to use infix notation, with either $\cdot$ or an empty string rather than functional notation. This makes some of the equations above look more familiar.

$$f(x^m, x^n) = x^{m+n},$$

for example, becomes

$$x^m \cdot x^n = x^{m+n}$$

or just

$$x^m x^n = x^{m+n}.$$

Also, because of the generalised associativity property, we don't need parentheses to indicate the order of operations, so we can write expressions like

$$xyx^{-1}y^{-1}$$

without specifying which of the five possible orders of operations are intended. When using this notation there are two different conventions for the identity element. Some authors use 1 and some use e.

This notation is less cumbersome than functional notation, and much less cumbersome than the relational notation from the set theory chapter, but it can be confusing in two situations. One is where we have multiple semigroups, each with its own binary operation. The other is where symbols like $\cdot$ or 1 have previously established meanings which conflict with the usage here, as when discussing the integers with their additive structure.

Regular languages

An important category of languages is the regular languages. These can be characterised in a variety of ways, via regular grammars, finite state automata, regular expressions, or syntactic monoids. It's important to understand all of these points of view because often a problem which is hard to solve using one description is easy in another.

To simplify things all of our examples in this chapter will be languages where the tokens are single characters and we'll write lists of tokens as strings. When writing grammars each terminal symbol will have only a single token, i.e. character, and will be denoted by that character. The characters `:`, `"` and `|` which have a special meaning in our language for describing languages, will not be tokens in any of our example languages, nor will any whitespace characters. Non-terminal symbols will always be denoted by a string more than one character long. These aren't limitations imposed by the theory, just ways of making the examples easier for you to read.

List of characters are strings. Because all tokens in our examples are characters all lists of tokens are strings. Strings are more familiar than lists of tokens so I'll often refer to them as strings when doing examples. I may occasionally make the mistake of referring to strings rather than lists of tokens in the general theory as well.

Regular grammars

Definitions of regular grammars vary but usually it's fairly easy to convert a grammar satisfying one definition to one satisfying another which generates the same language. I'll use the following definitions.

A left regular grammar is a phrase structure grammar where

- each alternate in the rule for the start symbol is a single non-terminal symbol,
- the start symbol never appears on the right hand side of a rule, and
- each alternate in the rule for any other non-terminal symbol is either empty or is a single non-terminal symbol followed by a single terminal symbol.

A right regular grammar is a phrase structure grammar where

- each alternate in the rule for the start symbol is a single non-terminal symbol,
- the start symbol never appears on the right hand side of a rule, and
- each alternate in the rule for any other non-terminal symbol is either empty or is a single terminal symbol followed by a single non-terminal symbol.

A simple, but not terribly useful, language is the language of any number of x's followed by any number of y's. Here "any number" includes zero, so the empty string, for example, belongs to this language. A left regular grammar for this language is

```
weird : xxyy

xxyy  : | xxyy "y" | xx "x"
xx    : | xx "x" | error "y"
error : error "x" | error "y"
```

The symbols xxyy and xx have the empty string as a possible expansion. The symbols start and error do not. The symbol error is not actually capable of generating any strings. Whenever we expand it we get another error symbol. The symbol xx can generate a string with any number of x's, including zero. The symbol xxyy can generate any string with x's followed by y's.

A right regular grammar for the same language is

```
weird : xxyy

xxyy  : | "x" xxyy | "y" yy
yy    : | "y" yy | "x" error
error : "x" error | "y" error
```

A more complicated, but more interesting, example is the language of decimal representations of integers, normalised in the usual way, i.e. there are no leading zeroes except for the integer 0, there is at most one leading $-$ sign, no $+$ sign, and there is no -0. We can write a right regular grammar for this language as follows:

```
integer : zero | pos_int | neg_int

zero    : "0" empty
empty   :
neg_int : "-" pos_int
pos_int : "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
digits  : | "0" digits
        | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
```

and a left regular grammar for the same language is

```
integer : zero | nonzero

zero    : empty "0"
nonzero : nonzero "0" | nonzero "1" | nonzero "2" | nonzero "3"
        | nonzero "4" | nonzero "5" | nonzero "6" | nonzero "7"
        | nonzero "8" | nonzero "9" | head "1" | head "2"
        | head "3" | head "4" | head "5" | head "6" | head "7"
        | head "8" | head "9"
head    : | empty "-"
empty   :
```

Both of these languages have both a left regular grammar and a right regu-

lar grammar. In fact every language which has a left regular grammar also has a right regular grammar and vice versa, but we're not yet in a position to prove this.

Closure properties

We can construct complicated languages from simpler languages in a variety of ways. It's useful to be able to construct a grammar for the more complicated language from grammars for the simpler languages from which it's built.

Unions

Languages are sets of lists. As sets it makes sense to talk about unions, intersections and relative complements. The union of two languages is again a language, as is the intersection or relative complement. Given left regular grammars for a pair of languages can we give a left regular grammar for their union? For the intersection? For the relative complement? The answer in each case is yes, but this is only easy to do for the union. We just need to create a new rule for the start symbol, which includes all the alternates for the start symbols of the original two languages, and copy all the rule for the other symbols, changing names if necessary to avoid duplicates. So the union of the two languages above has the grammar

```
union   : xxyy | zero | pos_int | neg_int

xxyy    : | xxyy "y" | xx "x"
xx      : | xx "x" | error "y"
error   : error "x" | error "y"
zero    : "0"
neg_int : "-" pos_int
pos_int : "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
digits  : | "0" digits
        | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
```

Of course the same remarks apply to right regular grammars as well. The union of two languages with right regular grammars has a right regular grammar. Also, the construction above is easily adapted to the union of finitely many languages.

Concatenation

We can also define a language whose members are the concatenation of a member of the first language and a member of the second. As with the union, we can construct a grammar for this new language from grammars for the two old languages by a purely mechanical procedure, though this time it's rather more complicated. It may be easiest to understand the procedure through examples. Consider, then, the language consisting of integers followed by some number of `x`'s and then some number of `y`'s.

We can start from our right regular grammar for the integers. Instead of an empty string at the end the of the input we should now have a string in the xy language, so we replace the empty alternatives in the right regular grammar for integers with the start symbol in the right regular grammar for the xy language.

```
intxxyy : zero | pos_int | neg_int

zero    : "0"
neg_int : "-" pos_int
pos_int : "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
digits  : weird | "0" digits
        | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
```

In this case there was only empty alternative, in the rule for the `digits` symbol. I haven't added in the rules from the xxyy grammar yet because we have a problem. `weird` is a non-terminal symbol and `digits` is also a non-terminal symbol, but the alternatives in a rule for a non-terminal symbol in a right regular grammar should be empty or a terminal followed by a non-terminal. As a first step to fixing this we need to replace `weird` by

its possible expansions.

```
digits  : xxyy | "0" digits | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits | "7" digits
        | "8" digits | "9" digits
```

There was in fact only one alternate, namely `xxyy`. This also isn't suitable as an alternate in a rule for a non-terminal symbol so we need to replace it by its possible expansions.

```
digits  : | "x" xxyy | "y" yy | "0" digits
        | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
```

Now we have a rule of the required form. We need to include more rules from the right regular grammar for the xy language so we can expand the symbols `xxyy` and `yy`. The complete grammar for the concatenation language is

```
intxxyy : zero | pos_int | neg_int

zero    : "0"
neg_int : "-" pos_int
pos_int : "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
digits  : | "x" xxyy | "y" yy | "0" digits
        | "1" digits | "2" digits | "3" digits
        | "4" digits | "5" digits | "6" digits
        | "7" digits | "8" digits | "9" digits
xxyy    : | "x" xxyy | "y" yy
yy      : | "y" yy | "x" error
error   : "x" error | "y" error
```

We can also construct a left regular grammar for this concatenation language from the left regular grammars for the integer language and the xy language. This time we start from the left regular grammar for the xy language and replace the empty alternates with the start symbol for the integer language.

```
intxxyy : xxyy

xxyy    : integer | xxyy "y" | xx "x"
xx      : integer | xx "x" | error "y"
error   : error "x" | error "y"
```

This grammar is not of the required form though so we need to replace `integer` by its possible expansions. Those rules will still not be be of required form, so we replace those replacements. The new rules for `xxyy` and `xx` are then

```
xxyy    : zero | nonzero | xxyy "y" | xx "x"
xx      : zero | nonzero | xx "x" | error "y"
```

Those rules will still not be be of required form, so we replace those replacements. The new rules for `xxyy` and `xx` are then

```
xxyy    : empty "0" | nonzero "0" | nonzero "1" | nonzero "2"
        | nonzero "3" | nonzero "4" | nonzero "5" | nonzero "6"
        | nonzero "7" | nonzero "8" | nonzero "9" | head "1"
        | head "2" | head "3" | head "4" | head "5" | head "6"
        | head "7" | head "8" | head "9" | xxyy "y" | xx "x"

xx      : empty "0" | nonzero "0" | nonzero "1" | nonzero "2"
        | nonzero "3" | nonzero "4" | nonzero "5" | nonzero "6"
        | nonzero "7" | nonzero "8" | nonzero "9" | head "1"
        | head "2" | head "3" | head "4" | head "5" | head "6"
        | head "7" | head "8" | head "9" | xx "x" | error "y"
```

The full grammar is then

```
intxxyy : xxyy

xxyy    : empty "0" | nonzero "0" | nonzero "1" | nonzero "2"
        | nonzero "3" | nonzero "4" | nonzero "5" | nonzero "6"
        | nonzero "7" | nonzero "8" | nonzero "9" | head "1"
        | head "2" | head "3" | head "4" | head "5" | head "6"
        | head "7" | head "8" | head "9" | xxyy "y" | xx "x"

xx      : empty "0" | nonzero "0" | nonzero "1" | nonzero "2"
        | nonzero "3" | nonzero "4" | nonzero "5" | nonzero "6"
```

```
        | nonzero "7" | nonzero "8" | nonzero "9" | head "1"
        | head "2" | head "3" | head "4" | head "5" | head "6"
        | head "7" | head "8" | head "9" | xx "x" | error "y"
error   : error "x" | error "y"
head    : | empty "-"
empty   :
```

The general procedure constructing a grammar for the concatenation language from the grammars for a pair of languages is as follows.

- Rename symbols to avoid name conflicts between the two grammars. Optionally rename other symbols for clarity.
- If we're constructing a left regular grammar start from a left regular grammar for the right element of the pair of languages. If we're constructing a right regular grammar start from a right regular grammar for the left element of the pair of languages.
- Replace all empty alternates with all alternates for all alternates of the start symbol in the other language.
- Add all rules from the other language other than the one for its start symbol.

As a further example I'll construct a right regular grammar for the concatenation of the xy language with itself. This time we'll need the renaming step mentioned above. We take two copies of the the right regular grammar for the xy language.

```
weirdl : xxyyl

xxyyl  : | "x" xxyyl | "y" yyl
yyl    : | "y" yyl | "x" errorl
errorl : "x" errorl | "y" errorl
```

and

```
weirdr : xxyyr

xxyyr  : | "x" xxyyr | "y" yyr
yyr    : | "y" yyr | "x" errorr
errorr : "x" errorr | "y" errorr
```

We're constructing a right regular grammar so we start from the grammar

for the left language.

```
weirdl : xxyyl

xxyyl  : | "x" xxyyl | "y" yyl
yyl    : | "y" yyl | "x" errorl
errorl : "x" errorl | "y" errorl
```

We then replace both empty alternates with `weirdr` and then replace that with `xxyyr` and then that with `| x xxyyr | y yyr`.

```
weirdl : xxyyl

xxyyl  : | "x" xxyyr | "y" yyr | "x" xxyyl | "y" yyl
yyl    : | "x" xxyyr | "y" yyr | "y" yyl | "x" errorl
errorl : "x" errorl | "y" errorl
```

Then we add in rules from the other language.

```
weirdl : xxyyl

xxyyl  : | "x" xxyyr | "y" yyr | "x" xxyyl | "y" yyl
yyl    : | "x" xxyyr | "y" yyr | "y" yyl | "x" errorl
errorl : "x" errorl | "y" errorl
xxyyr  : | "x" xxyyr | "y" yyr
yyr    : | "y" yyr | "x" errorr
errorr : "x" errorr | "y" errorr
```

It's important to understand which language we've just constructed a grammar for. A string is in this language if and only if it is the concatenation of two strings in the xy language. Those two strings could be the same but they don't have to be. The question of whether we can construct a grammar for the language of two repetitions of the same string is one we'll return to later.

Once we know how to construct a grammar for the concatenation of two languages we can construct a grammar for the concatenation of finitely many, by considering it at as a repeated concatenation.

Kleene star

Given a left or right regular grammar we can, using the techniques of the preceding section, construct, for each positive number n, a grammar for the language whose members are concatenations of n members of the original language. Of course we can can also do this for $n = 0$. In this case the language consists of only the empty list and it has the grammar

```
start : empty

empty :
```

We can also construct a grammar for the language of concatenations of between m and n members of a language, for natural numbers m and n, using the union construction earlier. What's less obvious is that we can construct a grammar for the language of concatenations of arbitrarily many members of a language.

Given a language the Kleene star of the language is set of all lists which can be found by concatenating arbitrarily many members of the language. This includes the empty list. The Kleene plus of the language is the set of lists which can be obtained by concatenating an arbitrary positive number of members of the language. If the original language had the empty list as a member then its Kleene star and Kleene plus are the same. If not then the Kleene star has the empty set as a member while the Kleene plus does not, but otherwise they are the same.

Given a regular grammar for a language we can find a regular grammar for its Kleene plus by looking through its rules for occurrences of the empty list and then adding alternates to any such rules. The alternates to be added are the alternates of alternates of the start symbol. To get a grammar for the the Kleene star we can apply our union construction discussed earlier to the Kleene plus grammar and the grammar given above for the language with just the empty list.

Applying the construction above to the xy language gives the following right regular grammar for its Kleene plus

```
weird : xxyy

xxyy  : | "x" xxyy | "y" yy
```

```
yy    : | "y" yy | "x" error | "x" xxyy | "y" yy
error : "x" error | "y" error
```

and the following grammar for its Kleene star

```
weird : xxyy | empty

xxyy  : | "x" xxyy | "y" yy
yy    : | "y" yy | "x" error | "x" xxyy | "y" yy
error : "x" error | "y" error
empty :
```

The constructions above are meant to show that regular grammars can be constructed for these languages. They do not attempt to find efficient grammars for them. For example, the Kleene star of the xy language is just the set of all lists of `x`'s and `y`'s. A much simpler grammar for this language is

```
ksxy : xyxy

xyxy : | "x" xyxy | "y" xyxy
```

Reversal

The reversal of a language is simply the set of the reversals of its members, where the reversal of a list is the list with the same items in reverse order. Given a left regular grammar for a language we can easily construct a right regular grammar for its reversal and vice versa. We just take the alternates in the rules for the grammar and reverse the order of the symbols. Here, for example, is a right regular grammar for the reversed integers, constructed from the left regular grammar for the integers.

```
integer : zero | nonzero

zero    : "0" empty
nonzero : "0" nonzero | "1" nonzero | "2" nonzero | "3" nonzero
        | "4" nonzero | "5" nonzero | "6" nonzero | "7" nonzero
        | "8" nonzero | "9" nonzero | "1" head | "2" head
```

```
        | "3" head | "4" head | "5" head | "6" head | "7" head
        | "8" head | "9" head
head    : | "-" empty
empty   :
```

What's less clear is how to construct a left regular grammar for the reversal from a left regular grammar for the original language or a right regular grammar for the reversal from a regular regular grammar for the original. This is a question we'll return to later.

Finite state automata

What I'm going to call a finite state automaton is more typically called a non-deterministic finite state automaton. Deterministic finite state automata, which will be considered in the next section, are a special case. I won't use the word non-deterministic except for emphasis. Unless a finite state automaton is specifically stated to be deterministic you should not assume that it is. Most other authors follow the reverse convention, assuming finite state automata are deterministic unless specifically allowed to be non-deterministic.

Non-deterministic finite state automata

To specify a finite state automaton we need the following:

- A set A of tokens,
- A finite set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

The interpretation of the ternary relation T is that $(r, a, s) \in T$ if the automaton can transition to the state s when it reads an a while in state r. The automaton must start in one of the states in I. If it's in one of the states in F at the end of the input then it halts successfully. It halts unsuccessfully if it is in a state not in F at the end of the input, or if it never reaches the end of the input because it reads a token in a state for which there is no transition allowed by T. As with all the other forms of non-deterministic calculation

we consider in this module the computation as whole is consider successful if some computational path is successful, even if others are not. In this case the finite state automaton is said to recognise the input. The set of lists of tokens recognised by a finite state automaton is said to be language recognised by it.

The non-determinism has two sources, the choice of initial state and the choice of the next state depending on the current state and the token just read. In most examples I has only one member and the first source of non-determinism is theoretical rather than real. Allowing multiple initial states is useful for the theory though, and sometimes in examples.

There is a traditional way of drawing diagrams for finite state automata, with directed graphs whose vertices are the states and whose edges indicate the allowed transitions, labelled to show which tokens allow that transition. The vertices in F are doubly circled. Those in $S \setminus F$ are singly circled. Vertices in I are indicated by unlabeled incoming arrows which don't come from any vertex. The accompanying diagrams show finite state automata which recognise the xy language and the language of integers.

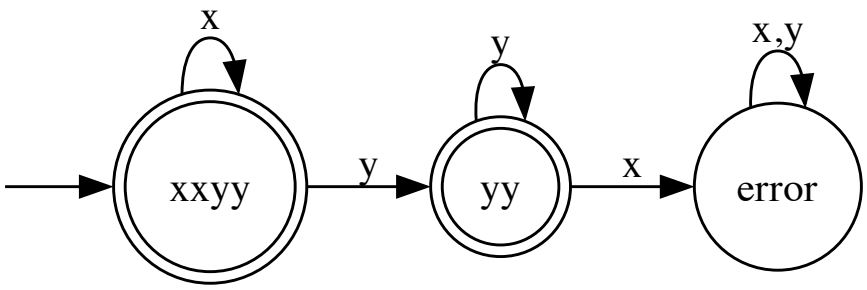


Figure 57: A finite state automaton for the the xy language

You may have noticed a similarity between these finite state automata and the right regular grammars for these languages. In fact it's possible to construct a finite state automaton from a right regular grammar by a purely mechanical procedure. The set A of tokens is the same as for the grammar. The set S of possible states is the set of non-terminal symbols of the grammar, except for the start symbol. The set I is the set of alternates for the start symbols. The set F is the set of non-terminals for which the empty list is an alternate. The set T consists of those (r, a, s) for which as is an alternate for

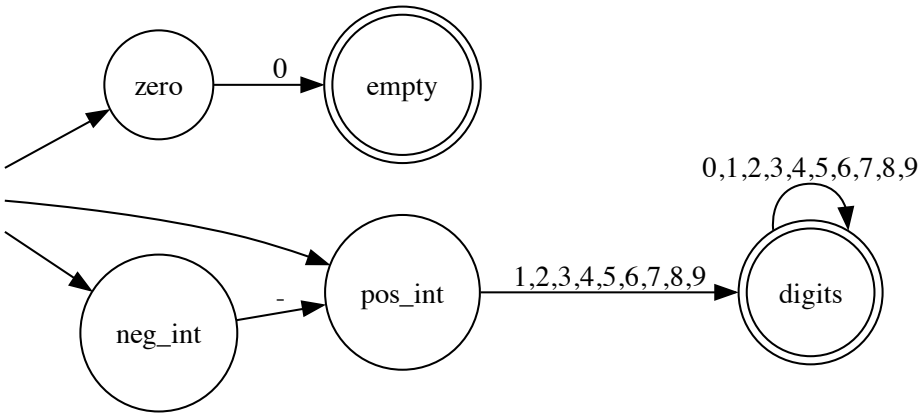


Figure 58: A finite state automaton for the the integer language

r.

So every language which has a right regular grammar is recognised by a finite state automaton.

Given a finite state automaton for a language we can easily construct a finite state automaton for the reverse language. A and S are unchanged. The roles of I and F are reversed. The transition relation for the reversed grammar consists of those triples (r, a, s) for which (s, a, r) belongs to the transition relation for the original language. If we have a left regular grammar for a language then we can find a right regular grammar for the reversed language, use it to construct a finite state automaton for the reversed language, and then use the construction above to construct a finite state automaton for the doubly reversed language, which is the original language. So every language which has a left regular grammar is recognised by a finite state automaton.

Deterministic finite state automata

A finite state automaton is called deterministic if the set of initial states has at most one member and for any state r and token a there is at most one allowed transition, i.e. at most one $s \in S$ such that $(r, a, s) \in T$. A deterministic finite state automaton therefore has at most computational path.

It is sometimes useful to strengthen the requirement of at least one start

state and at least one allowed transition in each state for each input token to exactly one start state and exactly one transition. The advantage of this is that it prevents the automaton from halting before it has read all of its input. I will call such automata strongly deterministic.

Our automaton above for the xy language is deterministic, and in fact strongly deterministic. Our automaton for the language of integers is not deterministic, since it has multiple initial states. A more common reason for a finite state automaton to be non-deterministic is the existence of multiple allowed transitions from a particular state on a particular input but this automaton happens not to have that problem.

Deterministic finite automata might seem more useful computationally than non-deterministic ones. This is only partly true. Other things being equal it's easier to work with a deterministic finite state automaton than a non-deterministic one, but other things are rarely equal. Often the simplest deterministic finite state automaton for a given language is much larger than the simplest non-deterministic one and it is therefore more efficient to accept the complications of non-determinism. It is nonetheless important, at least for theoretical purposes, to know that any language recognised by a finite state

Earlier I discussed how to take a finite state automaton which recognises a language and construct from it a finite state automaton which recognises the reversed language. This construction was simple, but it doesn't generally construct deterministic finite state automata from deterministic finite state automata. automaton is recognised by some deterministic finite state automaton.

We've already discussed how to simulate non-deterministic computations with deterministic ones. In general this is done with trees whose branches represent computational paths. The computational path describes not just the current state of the computation but also how it was arrived at. For finite state automata this is overkill. The future evolution of the computation, including whether it can terminate successfully, depends only on the current state, so we only need to keep track of the possible states the automaton could be in at each point in the input.

To illustrate this, consider the non-deterministic finite state automaton for the integers given earlier, and consider the input `-17`. Initially, i.e. before

any tokens are read, we are in one of the states `zero`, `neg_int`, or `pos_int`. We then read the token `-`. There are no transitions from the states `zero` or `pos_int` for this token and there is only one from `neg_int` so the only surviving computational path leaves us in `pos_int`. The next token is `1` and there is only one allowed transition from there so next we find ourselves in `digits`. Reading a `7` there leaves us in `digits`. At this point the input ends. We are in an accepting state so the computation is successful.

At each point in the input there is a set of states the computation could be in. This is initially the set I of initial states. The computation succeeds if one of the states it could be in at the end of the input is accepting, i.e. if the set of possible states has non-empty intersection with F. For each input token and set of states the system could be in before reading it we can compute the set of states it could be in after reading it by checking the allowed transitions for that token for each state.

The considerations above suggest the following power set construction. Given a non-deterministic finite state automaton described by a set of tokens A, a set of states S, a set of initial states I, a set of accepting states F and a transition relation T we construct a deterministic finite state automaton with the same set of tokens A, a set of states S', a set of initial states I', a set of accepting states F' and a transition relation T' according to the following rules.

- $S' = PS$, the power set of S.
- $I' = \{I\}$, the set with a single element, which is I.
- $F' = \{B \in PS : B \cap F \neq \emptyset\}$, the set of subsets of S whose intersection with F is non-empty.
- $T' = \{(B, a, C) \in PS \times A \times PS : C = \{s \in S : \exists r \in B : (r, a, s) \in T\}\}$. In other words C is the set of states to which there is an allowed transition on the input token a from a state in B.

This is indeed a deterministic finite state automaton because its set of initial states has only one element and for any $B \in S'$ and $a \in A$ there is only one $C \in S'$ such that $(B, a, C) \in T'$. At every point in the input the state of this automaton is the set of states the origin non-deterministic automaton could be in at the same point in the input. The deterministic automaton terminates successfully if and only if the non-deterministic one could terminate successfully. So they recognise the same language.

The construction above is called the power set construction, because the state space for the constructed automaton is the power set of the state space of the original automaton.

At this point I should give you an example of the construction but it's hard to find reasonable examples. Our automaton for the xy language is already deterministic. We could still apply the power set construction to it, obtaining a new automaton with eight states, but there's no point. Our finite state automaton for the integer language is genuinely non-deterministic so the power set construction does serve a purpose for it, but it gives us an automaton with 32 states. That's certainly implementable on a computer but its diagram wouldn't fit on a single page.

Closure properties

Earlier we discussed set operations for languages generated by regular grammars. More precisely, I showed that the union of two such languages is such a language, but I didn't answer the question for intersections or relative complements. For languages recognised by finite state automata I'll answer the question for intersections and relative complements, but not for unions. It's actually fairly easy to answer the question for unions as well, but it's unnecessary, as we'll see later.

Intersection

To construct a finite state automaton for the intersection of two languages from finite state automata for each language individually we just need to keep track of what states those two automata could be in at any point. To be more precise, suppose the two automata have the same set of tokens A and have sets of states S_1 and S_2, sets of initial states I_1 and I_2, sets of accepting states F_1 and F_2, and transition relations T_1 and T_2. We construct a finite state automaton with the same set of tokens A, a set of states S, a set of initial states I, a set of accepting states F and a transition relation T which recognises those lists which are recognised by both these automata as follows.

- $S = S_1 \times S_2$
- $I = I_1 \times I_2$
- $F = F_1 \times F_2$

- $T = \{((r_1, r_2), a, (s_1, s_2)) \in S \times A \times S : (r_1, a, s_1) \in T_1 \wedge (r_2, a, s_2) \in T_2\}$.

At every point in the input this automaton can be in the state (s_1, s_2) if and only if the first automaton can be in the state s_1 and second can be in the state s_2. It therefore can reach an accepting state at the end of the input if and only if both the original automata could.

So the intersection of two languages recognised by finite state automata is a language recognised by a finite state automaton.

Relative complements

It might seem obvious how to modify this construction for relative complements. We just need to replace accepting states by rejecting for one of the automata, right? This isn't completely wrong, but it's not completely right either. For one thing, it's possible for a finite state automaton to halt unsuccessfully before reaching the end of its input, if there are no allowed transitions for the symbol just read from the current state. For another, the fact that a non-deterministic automaton can end up in a rejecting state at the end of the input doesn't mean the input must be rejected, since some other set of choices for the initial state or transitions might leave it in an accepting state. Neither of these things can happen though if the finite state automaton is one which was constructed by the power set construction though, since those are always strongly deterministic, so if we first apply the power set construction to our finite automata and then the naive version of the relative complement construction described earlier it will work.

So the relative complement of two languages recognised by finite state automata is a language recognised by a finite state automaton.

Regular expressions

Regular expressions are both an important theoretical concept in computing and an important practical tool in programming. These two meanings for regular expressions are not quite the same though. For theoretical purposes it's convenient to have a minimally expressive syntax for regular expressions. The fewer ways to construct a regular expression the easier it is to prove their properties. For practical programming it's convenient to have a maximally expressive syntax, to make it easier to write simple

regular expressions for simple tasks. To complicate matters even further, most regular expression libraries provide not just syntactic sugar to make writing regular expressions easier but also extensions which increase their power as a computational tool. That may sound good if you're a programmer but it means that some of the statements I'll make below about what regular expressions can and can't do are simply untrue when applied to regular expressions as understood by those libraries. Since this module is concerned with the theory of computation rather than practical programming I will give the minimalist version but I will briefly mention the IEEE standard regular expressions understood by most libraries.

The basic operations

A language is called regular if it can be build from finite languages by the operations of union, concatenation and Kleene star. More precisely, suppose A is a finite set of tokens. Let F be the set of finite languages with tokens in A, i.e. the set of finite sets of lists of items in A. Let S be the set of all sets of languages defined by:

- $F \in S$.
- For all $R \in S$ if $L_1 \in R$ and $L_2 \in R$ then $L_1 \cup L_2 \in R$.
- For all $R \in S$ if $L_1 \in R$ and $L_2 \in R$ then $L_1 \circ L_2 \in R$, where $L_1 \circ L_2$ is the set of lists which are concatenations of a list in L_1 and a list in L_2
- If $R \in S$ and $L \in R$ then $L^* \in R$, where L^* the set of all concatenations of arbitrarily many members of L.

Then the set of regular languages for the set of tokens A is $\cap S$.

The notation $\circ$ for concatenation is unfortunate since it suggests composition but is in fact unrelated to it.

Intuitively, regular languages are built from finite languages using union, concatenation and Kleene star and are only those languages which can be built in a finite number of steps of those three types from finite languages.

Regular expressions are a notation for describing how a language is built from a set of finite languages using those components. There are a few different notations for regular expressions. I'll use one which is a subset of the IEEE notation. This has the limitation that it only works when the tokens are individual characters, but that's the case of most practical interest.

In IEEE regular expressions characters represent themselves, except that a few characters are special. To represent a special character we need to place a backslash \, before it. The special characters include \ itself, the parentheses (and), used for grouping, the vertical bar |, used for the union operation, and the asterisk *, used for the Kleene star operation. There is no special character for concatenation. Concatenation is indicated by concatenation.

There is exactly one regular language which cannot be expressed in this notation: the empty language. The IEEE standard has no way of denoting this language but I'll use $\emptyset$ for it.

Examples

It is probably easier to understand this via examples. `x*` is a regular expression for the language consisting of arbitrarily many copies of the character `x`. Similarly `y*` is a regular expression for arbitrarily many `y`'s. Then `x*y*` is a regular expression for arbitrarily many `x`'s followed by arbitrarily many `y`'s. In other words, `x*y*` is a regular expression for the xy language considered earlier.

Arbitrarily many includes zero, so the empty string is an element of this language. If we wanted to ensure that there is at least one `x` and at least one `y` we would have to use the regular expression `xx*yy*`, i.e. a single `x`, followed by arbitrarily many `x`'s, followed by a single `y`, followed by arbitrarily many `y`'s. This is, of course, a different language from the one in the preceding paragraph.

As our next example let's try to build a regular expression for the language of integers. It will be easiest to do this in stages. `0|1|2|3|4|5|6|7|8|9` is regular expression for the language of single digits. We don't need parentheses for grouping here because union is an associative operation. To get strings of digits we apply Kleene star: `(0|1|2|3|4|5|6|7|8|9)*`. The parentheses are needed for precedence, specifically to express the fact that this arbitrarily many digits rather than either a non-`9` or arbitrarily many `9`'s, which would be `0|1|2|3|4|5|6|7|8|(9*)`. The language described by the regular expression `(0|1|2|3|4|5|6|7|8|9)*` includes the empty string and also `007`. The former is not an integer and the latter is an integer but doesn't satisfy the normalisation conditions we imposed

earlier to make sure each integer has a unique representation. Both of these problems can be fixed by making sure there's a non-zero digit before the `(0|1|2|3|4|5|6|7|8|9)*`. A regular expression for non-zero digits is `1|2|3|4|5|6|7|8|9` so we're led to

```
(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*.
```

This is a regular expression for the language of positive integers. To allow negative integers we concatenate the regular expression `|-` with this regular expression to get

```
(|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*.
```

A `|-` matches either the empty string or a single `-`. We now have a regular expression which matches all nonzero integers. A regular expression which matches zero is just `0`. The following regular expression therefore matches all integers:

```
(0|(|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*).
```

Towards the end of the example I started using the standard term "match" for the situation where a string is a member of the language described by a regular expression. It's quite convenient and I'll use it without comment from now on.

From regular expressions to grammars

Converting a regular expression to a left or right regular grammar for the language it recognises might seem difficult, and doing it efficiently is indeed difficult, but as long as we just want some regular grammar and don't care about efficiency it's easy.

First of all, it's easy to write down a grammar for a single character. For example, here is a grammar for the language whose only string is a single `x`:

```
start : one_x

one_x : "x" empty
empty :
```

This is a right regular grammar. A left regular grammar for the same language would look the same, except the expansion for `one_x` would be `empty x`.

We've already seen how to construct regular grammars for the union, concatenation or Kleene star of languages from regular grammars for the original languages though, and regular expressions are built up from grammars for a single character or the empty string using those three operations so in principle we know how to construct either a left or right grammar for the language described by any regular expression. This procedure will generally give us an unnecessarily complicated grammar for the language, but it will give us a grammar. It follows that every regular language can be described by a left regular grammar and by a right regular grammar.

Regular expressions from automata

Any language which can be recognised by a finite state automaton can be described by a regular expression. This is proved by induction on the number of states. To make the induction work though we first need to generalise our notion of finite state automata. The automata we've considered read single tokens and make a state transition based on the token read. For each pair of states r and s there is a set, possibly empty, of tokens which all a transition from r to s. We can represent this set of tokens by a regular expression, consisting of those tokens with |'s between them.

A generalised finite state automaton will be one where for each pair of states r and s there is a regular expression such that if the automaton reads a list of tokens matching that regular expression while in the state r then it can transition to the state s. Every finite state automaton is a generalised finite state automaton, where the regular expression is just the one described previously, i.e. the list of tokens separated by |'s. We can add two states to this automaton to create a new one which has only a single initial state and a single accepting state. To do this we demote the previous initial states and accepting states to ordinary states and add a new initial state and a new accepting state. We then allow transitions from the new initial state to the old ones where the regular expression is the one matching the empty list and transitions from the old accepting states to the new one, again with the regular expression being the one for the empty list. The new automaton can therefore go from the new initial state to one of the old ones without

reading any input, then proceed as before, arrive at one of the old accepting states at the end of its input, and then transition to the new accepting state without reading any further input. The states other than the initial and accepting states will be called intermediate states.

Suppose we have a generalised finite state automaton with a single initial state and a single accepting state and at least one intermediate state. We can construct another generalised finite state automaton which recognises the same language, also with a single initial state and a single accepting state, but with one intermediate state fewer, as follows.

We pick an intermediate state r. We want to remove r but some computational paths go through r so we will need to replace them with paths which don't. For each pair of other states q and s there are possibly paths which go from q to r and then from r to s. We can replicate the effect of those paths by unioning the existing regular expression for transitions from q to s with the concatenation of the regular expression for transitions from q to r with the one for transitions from r to s. This isn't quite enough though, since a computational path might stay at r for an arbitrary number of steps before moving on to s. So what we need to add to the regular expression for transitions from q to s is a concatenation of three regular expressions: the one for transitions from q to r, the Kleene star of the one for transitions from r to itself, and then the one for transitions from r to s. Once we've done this for all pairs q and s the state r is no longer needed and can be removed.

Removing intermediate states one after another we eventually reach the point where there are no intermediate states. We're left with just initial and accepting states. If we did the construction as described above then there is one of each and there are no allowed transitions from the initial state to itself or from the accepting state to itself or to the initial state. The only allowed transition is one directly from the initial state to the accepting state. Input will be accepted by this machine if and only if it matches the regular expression for that transition. In this way we've found a regular expression which matches precisely those inputs recognised by the original finite state automaton.

I'm not going to give an example of the construction above. The regular expressions it produces are horribly inefficient. The point of the construction is just to prove that every language recognised by a finite state automaton

is a regular language.

Reversal

One nice property of regular expressions is that given a regular expression for a language it's very easy to construct a regular expression for the reversed language. Unions and Kleene stars can be left unchanged and we just need to reverse the order of the concatenations. For example, a regular expression for the reversed integers is

```
(0|(0|1|2|3|4|5|6|7|8|9)*(1|2|3|4|5|6|7|8|9)(|-)).
```

Extended syntax

For practical purposes it's useful to have more operations available than just union, concatenation and Kleene star. We didn't include those extra operations in the definition to avoid needing to prove the corresponding closure properties of regular grammars.

In the IEEE standard `+` is used for Kleene plus, i.e. a concatenation with at least one member. `?` is used for at most one member, but possibly zero. Also explicit ranges are allowed, denoted by numbers in braces. For example `(0|1|2|3|4|5|6|7|8|9){3,5}` would indicate a string of at least three and at most five digits. Character ranges are also allowed, indicated by brackets, so three to five digits could also be represented as `[0-9]{3,5}`. Of course this requires `+`, `?`, braces and brackets to be special characters, which then have to be preceded by backslashes in order to represent themselves. There are a few other similar extensions. If you're using regular expressions for pattern matching, and you really should if you have to do pattern matching, then you should consult the documentation for whatever library you're using, both to see what extensions are available and to see which characters are special. Even if you will never use an extension you may need to know about it if it makes certain characters special and therefore requires you to precede them with backslashes.

The extensions above are a matter of syntactic convenience. They don't change the set of languages which can be represented; they just make it possible to represent some languages with shorter regular expressions. There are other extensions in many implementations which change the set of rep-

resentable languages. The new languages which these allow cannot be generated by regular grammars. None of what I say in this chapter about regular languages can be assumed to apply to the languages described using these extensions.

Regular languages

We've now seen how to go from a left or right regular grammar to a finite state automaton, from a finite state automaton to a deterministic finite state automaton, from there to a generalised finite state automaton, from there to a regular expression, and finally from a regular expression to a left or right regular grammar. At each step the language is unchanged. We can therefore conclude that the following sets of languages for a given set of tokens are all the same:

- the set of languages with a left regular grammar
- the set of languages with a right regular grammar
- the set of languages recognised by a finite state automaton
- the set of languages recognised by a deterministic finite state automaton
- the set of languages recognised by a generalised finite state automaton
- the set of languages described by a regular expression

The last of these was our definition of a regular language, but we could really have taken any of them as our definition.

The accompanying diagram describes all the constructions given above, with black arrows representing ways to get one description of a regular language from an other and the blue arrows representing ways to get a description of a regular language from one of its reversal.

This equivalence now allows us to answer many questions which were left unanswered in the sections from individual points of view.

I stated earlier, for example, that every language generated by a left regular grammar can also be generated by a right regular grammar and vice versa. We know this is true. In theory the proof is even constructive. We could take a left regular grammar, construct the corresponding finite state automaton, use the power set construction to construct a deterministic fi-

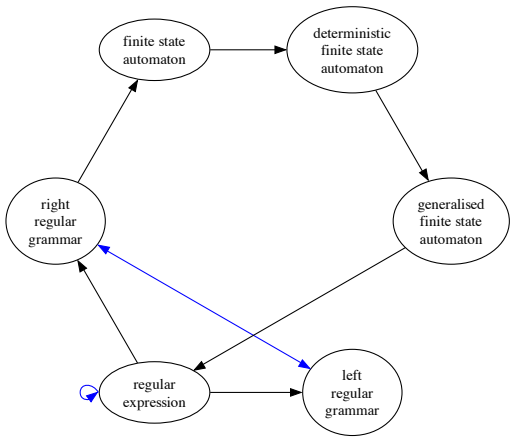


Figure 59: Descriptions of a language

nite state automaton, convert it to a generalised finite state automaton with a single initial and single accepting state, kill all of its intermediate states one by one, take the resulting regular expression, and then use it to find a right regular grammar. All the steps in this process can in principle be carried out in a purely mechanical way. You shouldn't ever do this, of course. The resulting grammar would be horrible.

We also considered closure of these sets of languages under various set operations. It was easy, for example, to see that the union of languages with a regular grammar has a regular grammar. We can now see that that's true of languages described by any of the three types of finite automata or by regular expressions. This would be easy to prove directly for regular expressions but is quite tricky to prove for deterministic finite state automata. On the other hand it was fairly straightforward to prove that the intersection of languages defined by such finite state automata is also defined by such an automaton but this is far from obvious for languages defined by regular grammars or regular expressions. It must be true though, since these are all different ways of describing the same set of languages.

Similarly, reversal is an easy process to describe in terms of regular expressions but it's far from clear how to take, for example, a left regular grammar and construct a left regular grammar for the reversed language, or to take a deterministic finite state automaton for a language and construct a deterministic finite state automaton for the reversed language. The equivalence

of all of these descriptions of regular languages shows that it must be possible though.

In general if you want to prove a fact about regular languages you should look for the description in terms of which this fact is easiest to prove. You can even mix them. If regular languages appear in both the hypotheses and conclusion of a theorem you want to prove you might find it convenient to use one characterisation for the hypotheses and another for the conclusion.

Pumping lemma

One consequence of the equivalence discussed in the preceding section is that we have six different ways to show that a language is regular:

- give a left regular grammar which generates its members
- give a right regular grammar which generates its members
- give a finite state automaton which recognises its members
- give a deterministic finite state automaton which recognises its members
- give a generalised finite state automaton which recognises its members
- give a regular expression which matches its members

That's nice, but we've seen zero ways so far of showing that a languages isn't regular. That's a rather serious gap and none of the descriptions we have are of much help here. We can hardly list all left regular grammars, for example, and check that none of them generate the language. In order to prove that a language isn't regular you need to identify a property which all regular languages share and then show that this language does not have that property. There are two properties which people use for this.

One of these is the subject of the Myhill-Nerode theorem, which we'll discuss in the next section. The other is that of the Pumping lemma, which I'll discuss in this section. The Myhill-Nerode theorem is better in nearly all respects than the Pumping lemma. It provides a necessary and sufficient condition for regularity while the Pumping lemma just provides a necessary condition and, although this is necessarily somewhat subjective, I find it much easier to use. There are two reasons to introduce the Pumping lemma anyway though. One is that it's more popular. If you ever see

someone outside of this module proving a language is not regular they will probably be doing so using the Pumping lemma so you should know what it is. The second reason is that there are two Pumping lemmas, one for regular languages and one for context free languages. It's the first of these that we're discussing in this chapter. The second will be discussed in the next chapter. The Myhill-Nerode theorem doesn't have such a nice generalisation to context free languages so we will need the second version of the Pumping lemma in order to prove that various languages are not context free in the next chapter. For this reason not only will I give a proof of the Pumping lemma for regular languages but I'll give one which, unlike the usual proof, generalises well to context free languages.

I haven't introduced a notation for concatenation yet. We'll be encountering a lot of them in the remainder of this chapter and it's convenient to have a notation for them. I'll take the simplest option and denote concatenation by concatenation. In other words, if u and v are lists of tokens then uv will be the list obtained by concatenating u and v, in that order. I'll also write u^n for concatenation of n copies of u.

The statement of the lemma

With those preliminaries out of the way we can proceed to the statement of the Pumping lemma, which is a bit weird. This will require some terminology. We say that a natural number p is a pumping length for a language L is for every member w of L of length at least p can write w as a concatenation of three lists a, b and c, in that order, i.e. $w = abc$, with the following properties:

- b is not the empty list,
- ab has length at most p, and
- for every natural number n the list ab^nc is a member of L.

Note that p depends on L but not on w. The pumping length is a property of the language, not any particular member.

A language is said to have the pumping property if it has a pumping length. The Pumping lemma says that every regular language has the pumping property. It does not say that every language with the pumping property is regular, and indeed that's not true.

There are really two pumping lemmas for regular languages, a left pumping lemma and a right pumping lemma. For some reason everyone seems to state the version above, but there's also a version which is identical except that it's bc which has length at most p.

Examples

Before giving a proof I'll do an example to show how the Pumping lemma can but used to show that a language isn't regular.

Earlier we considered a language whose members are all strings of the form some number of x's followed by some number of y's. This was a regular language. We know this because we've seen a left regular grammar for it, a right regular grammar for it, a deterministic finite state automaton for it, and a regular expression for it. Any one of these would suffice to prove that it is regular. We therefore can't expect to use the Pumping lemma to show that it's not regular. Consider, though, the language of strings which are some number of x's followed by the same number of y's. We can show, using the Pumping Lemma, that this language is not regular.

The proof is by contradiction. Suppose the language is regular. Then it has the pumping property, i.e. there is some pumping length p for this language. Let w be the string with p x's followed by p y's. It belongs to the language so it can be written as abc as in the definition of the pumping length. Because ab is of length at most p and occurs at the beginning of w both a and b must be strings of x's. Consider the string ac, thought of as ab^0c This is the case $n = 0$ of ab^nc. and so is a member of L. b is of positive length and consists solely of x's so by removing it we now have a string in the language with fewer than p x's followed by p y's. But the language is the language of strings where some number of x's is followed by the same number of y's, so this is impossible.

The name of the Pumping lemma comes from the possibility of taking n to be large, generating arbitrarily long language elements by a process of "pumping". We could have done that here but we didn't need to. Instead of lengthening our string to get a contradiction we shortened it.

This language, which we've just shown not to be regular, is a sublanguage of our original xy language, which we already knew to be regular. It follows that not every sublanguage of a regular language is regular.

The name of the Pumping lemma comes from the possibility of taking n to be large, generating arbitrarily long language elements by a process of "pumping". We could have done that here but we didn't need to. Instead of lengthening our string to get a contradiction we shortened it.

The Pumping Lemma is only every used in proofs by contradiction. The steps are always

- Assume the language is regular, so there is some pumping length p for it.
- Choose a list (string), depending on p.
- Show that for the chosen string and every non-empty segment in the first p tokens (characters) there is some number of repetitions which takes us outside the language.
- Conclude that the language wasn't really regular, since it doesn't satisfy the pumping property.

Note that we we negate a statement all universal quantifiers become existential and vice versa, e.g. the negation of "there is a p such that for all lists of length at least p there is a non-empty segment ..." is "For all p there is a list of length at most p such for all non-empty segments ...".

As another example, consider the language of balanced parentheses. For this language we choose a string with p ('s followed by p)'s. This is a member of the language of balanced parentheses. Any substring within the first p characters consists only of ('s. If we repeat that substring 0 times we get fewer ('s than)'s. The last) has no (to match it, so this string is not in the language, so we're done; the language of balanced parentheses is not regular. Here it was fairly easy to guess what string to chose. Often it's not.

Finite languages

Students occasionally get confused by one point about the Pumping lemma. Finite languages are always regular. The Pumping lemma appears to allow us to generate arbitrarily long strings in a regular language. How is this not a contradiction? The resolution of this seeming paradox is that the Pumping lemma only says something about sufficiently long strings, specifically those greater than the pumping length of the language. A finite language has a pumping length equal to the length of its longest string. There are

no strings w in the language with length longer than that so the statement about being able to split w into a, b and c with the given properties doesn't actually apply to any string and is vacuously true.

The proof of the lemma

Suppose L is a regular language. It must then have a left regular grammar. Let p be the number of non-terminal symbols in this grammar. I will show that p is a pumping length for L. Suppose $w \in L$ is of length m, which we assume is at least p. The parse tree for w is of a particularly simple form. The root has one child. Almost every other node has two children, one of which is a leaf with a terminal symbol and the other of which is a non-terminal symbol, also with two children. The one exception is that the non-terminal symbol which gets expanded to the empty list has no children and is therefore also a leaf.

Our parse tree has m leaves labelled by terminal symbols, each with a distinct parent, labelled by a non-terminal symbol other than the start symbol, and the one non-terminal symbol which is a leaf, for $m + 1$ non-terminals other than the start symbol. Since $m + 1$ is greater than p some non-terminal is repeated symbol. There may well be more than but there must be one within the last $p + 1$ symbols. We can take the segment of the tree between those symbols, including the one closest to the root and excluding the one farthest from the root, and repeat this as many times as we want to get parse trees for valid lists in the language. The part of the tree below the repeated segment is our a, the repeated part is b and the part above is c.

The accompanying diagrams illustrate this construction on the string `2023` in the integer language. The string `02` between `2` and `3` can be repeated arbitrarily many times. The parse tree for the original string is shown along with the trees for zero repetitions and two repetitions.

To get the version of the Pumping lemma where it's bc which is of length at most p we would apply the same construction to a right regular grammar.

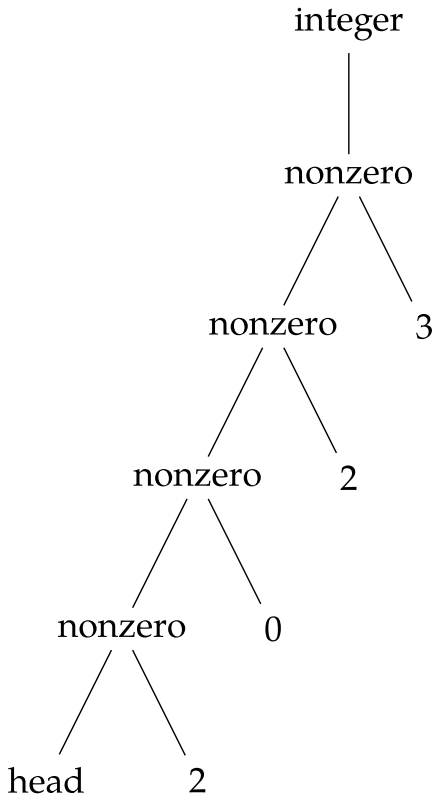


Figure 60: Parse tree for the string 2023

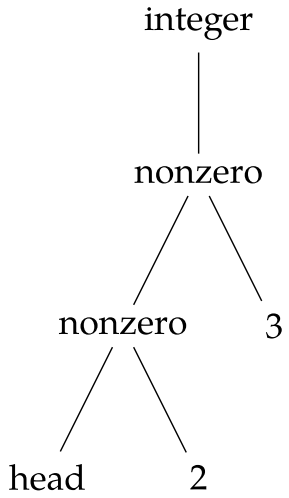


Figure 61: Parse tree for the string 23

The Myhill-Nerode theorem

One problem with the Pumping lemma is that it can be difficult to guess which w you need to take in order to find a contradiction. Another problem is that there are languages with the pumping property which are nonetheless not regular. There is a nice necessary and sufficient condition for regularity but it will require some preliminaries.

From languages to automata

From a language we can directly construct an automaton which recognises it, as described below. This automaton may or may not be finite.

We start with a set of tokens A and a language L, i.e. a subset of B, the set of all lists of members of A. Let ε be the empty list. We define a function f from B to PB by

$$f(w) = \{z \in B : wz \in L\}.$$

We call $f(w)$ the set of valid continuations of w, since $z \in f(w)$ if and only if reading z after reading w gives us a member of the language. Note that $\varepsilon \in f(w)$ if and only if $w \in L$. Also, $f(\varepsilon) = L$. Both of these statements follow from the fact that ε is the identity for B with the operation of concatenation. Let C be the range of f, i.e. the set of valid continuation sets.

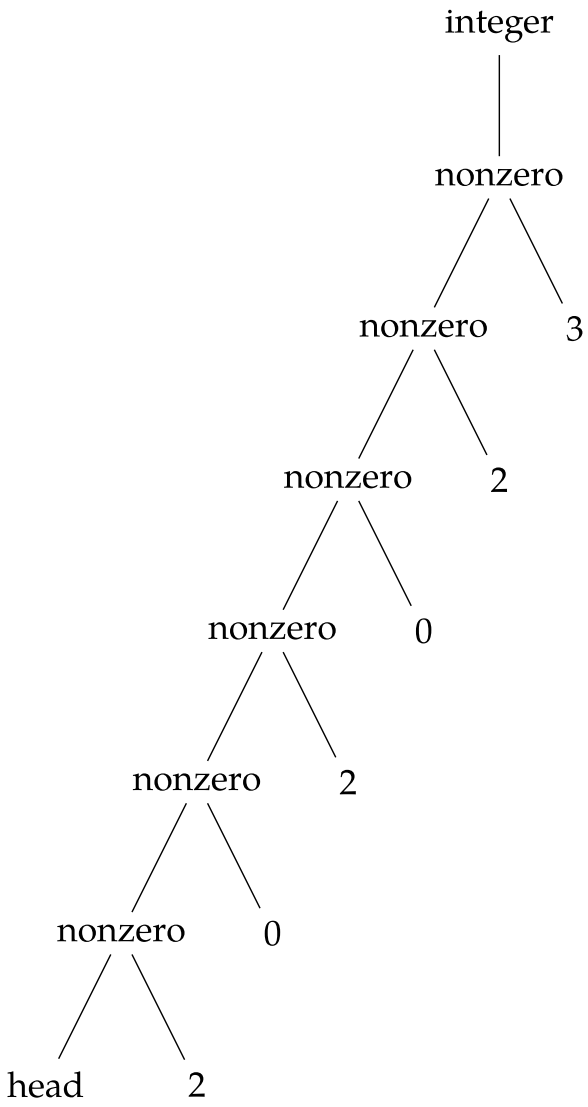


Figure 62: Parse tree for the string 202023

We define an equivalence relation on B by saying that u and v are equivalent whenever $f(u) = f(v)$. Let E be the set of equivalence classes. Let g be the function from B to E which takes each list to the equivalence class to which it belongs. Every equivalence class is the equivalence class of some list. One way to express this is to say that g is a surjective function, i.e. that if $P \in E$ then $P = g(u)$ for some $u \in B$. If $P = g(v)$ then u and v are equivalent, i.e. $f(u) = f(v)$, so $f(u)$ depends only on P and not on the particular u chosen. It is therefore legitimate to define $h(P) = f(u)$. h is a function from E to C. It was defined in such a way that $h(g(u)) = f(u)$ for all u, i.e. such that $f = h \circ g$. h is a injective because if $h(P) = h(Q)$ then $P = g(u)$ and $Q = g(v)$ for some u and v in B, but then $f(u) = f(v)$ so u and v are equivalent and so $g(u) = g(v)$, or in other words $P = Q$. h is surjective since if $R \in C$ then $R = f(w)$ for some $w \in B$ and then $R = h(g(w))$ and hence $R = g(P)$ for some $P \in E$.

Suppose $P \in E$ and $w \in B$. Let

$$R = \{z \in B : wz \in P\}$$

Now $P = f(u)$ for some $u \in B$ so

$$R = \{z \in B : wz \in f(u)\}$$

or

$$R = \{z \in B : uwz \in L\}$$

and therefore

$$R = f(uw).$$

$f(uw)$ is a member of C and g is a bijective function from E to C so there is a unique $Q \in E$ such that $R = g(Q)$, i.e. such that

$$g(Q) = \{z \in B : wz \in P\}.$$

We can therefore define a function t from $E \times B$ to E by

$$g(t(P, w)) = \{z \in B : wz \in P\}.$$

An alternate way to describe this is that $x \in t(P, w)$ if and only if there is some $u \in P$ such that $x = uw$.

Given any list $w = (a_1, a_2, \ldots, a_n)$ of tokens we can form the list of equivalence classes $(s_0, s_1, s_2, \ldots, s_n)$ where

$$s_0 = g(\varepsilon), \quad s_1 = t((a_1), s_0), \quad s_2 = t((a_2), s_1), \quad \cdots \quad s_n = t((a_n), s_{n-1}).$$

By induction we have

$$g((a_1, a_2, \ldots, a_j)) \in s_j$$

for all j and so, in particular

$$g(w) \in s_n.$$

Then

$$f(w) = h(s_n)$$

Now $w \in L$ if and only if $\varepsilon \in f(w)$, i.e. if and only if $\varepsilon \in h(s_n)$. Let

$$I = \{g(\varepsilon)\},$$

$$F = \{s \in E : \varepsilon \in h(s)\},$$

and

$$T = \{(r, a, s) \in E \times A \times E : s = t(s, (a))\}.$$

Then $(s_0 \in I)$, (s_j, a_j, s_{j+1}) for all $j < n$ and $w \in L$ if and only if $s_n \in F$. In other words, if we form the automaton whose state set is E, whose initial and accepting sets are I and F respectively and whose transition relation is T then this automaton recognises L. In particular if E is finite then we have a finite state automaton which recognises L. We'll see later that the converse is also true, that if there is a finite state automaton which recognises L then E is finite, but first it may be helpful to do an example.

An example

We can use the construction above to find a finite state automaton for the language of integers.

What are the equivalence classes of strings of the characters `0`, `1`, ..., `9`, and `-`?

- We always have the equivalence class of the empty string, whose continuation set is just the language. There is no other string whose continuation set has all integers as members so the empty string is the only member of this language.

- There is also the equivalence class of the string `0`. The only continuation of `0` is the empty string. There are no other strings whose only continuation is the empty string, so `0` is the only member of this equivalence class.
- We also have the equivalence class of `-`. The continuations are just the strings representing positive integers. There is no other string with the same continuation set so `-` is the only member of this equivalence class.
- There is also an equivalence class whose members are all non-zero integers. These all have all strings of digits as their continuations. That includes an empty string of digits.
- Finally, there is an equivalence class consisting of those strings with no continuations. These are the strings with some sort of syntax error, like `0--`.

These are all the equivalence classes. We've just seen that we can form a finite state automaton whose states correspond to the equivalence classes. The only initial state is the one corresponding to the equivalence class of the empty set. The accepting states are the ones for which the empty list is a valid continuation, which in this case is the class of `0` and the class of non-zero integers.

There are two reasonable ways to label these states. One is with the equivalence classes and the other is with the continuations. We saw in the last section that there is a bijective function, which we called h from equivalence classes to continuations, so either will work. I find it easier to understand the version with states labelled by continuations. There is one slightly tricky point. We have one state where the set of continuations is empty and one where the only member of the set of continuations is the empty list. We can't label both of them `empty`. I'll use that label for the second one, and the label `error` for the first one, since that's the state we're in if there has been a syntax error in the input. With these choices the finite state automaton for the integer language is the one with the accompanying diagram.

This finite state automaton is deterministic, and in fact strongly deterministic. This is not an accident. The construction from the previous section always gives a strongly deterministic automaton, and indeed gives one with

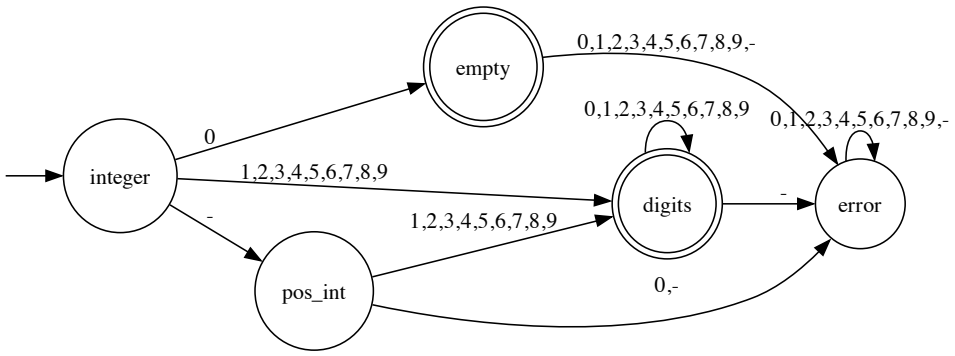


Figure 63: A strongly deterministic finite state automaton for the integers

as few states as possible.

The converse

We've seen that if the set E of equivalence classes is finite then there is a finite state automaton which recognises the language. In this section we'll see that the converse is also true. If a language is recognised by a finite state automaton then E is finite.

Suppose we have a finite state automaton with S as its set of states which recognises the language L. As usual, F will be the set of accepting states. As before I'll denote the set of all lists of members of A by B. For each w in B let $i(w)$ be the subset of S consisting of those states which the automaton can be in after reading w. There could be more than one member of $i(w)$ if the automaton is non-deterministic, and there could be none if it is not strongly deterministic. If it is strongly deterministic then $i(w)$ has exactly one member. Let D be the range of i.

If z is a continuation of w then wz is a member of the language and so must be accepted by the automaton, so there must be a computational path for z` from a member of $i(w)$ to a member of F. Conversely, if there is such a path then wz must be a member of the language, so z is a continuation of w. In particular the set $f(w)$ of continuations of w depends only on $i(w)$. So if we define j to be the function from the range of i in PS to C which takes a subset $H \subseteq S$ to the set of strings which can be accepted by the automaton from some state of H then $f = j \circ i$. Since we already have $f = h \circ g$ we have

$j \circ i = h \circ g$. Let $k = j \circ h^{-1}$. This makes sense since h was already shown to be bijective. Then $k \circ i = g$. k is a surjective from the range of i, which is a subset of PS, to E. S is finite, so PS is finite, so the range of i is finite, and therefore E is finite.

In fact we can be more precise. If there are n states then there are 2^n members of PS and so at most 2^n members of the range of i and then at most 2^n members of E. If the finite state automaton is strongly deterministic then every member of the range of i has a single member and there are only n such subsets of S, so we get the much stronger result that E has at most n members. In particular every strongly deterministic finite state automaton which recognises L has at least as many states as E has members. In an earlier section we constructed a strongly deterministic finite state automaton with exactly that many members. We can now see that that automaton is minimal, in the sense that it has the smallest possible number of states for a strongly deterministic automaton which recognises L.

We can now state one form of the Myhill-Nerode theorem, that if L is a language and E is the set of equivalence classes of lists with respect to L, equivalence being defined by saying that lists are equivalent if they have the same set of continuations, then L is regular if and only if E is finite.

The syntactic monoid

There are two other forms of the Myhill-Nerode theorem. One of these can be proved by applying the previous version to the reversed language. The reversed language is regular if and only if the original one is. In this second version of the Myhill-Nerode theorem the set which we need to be finite is the set of equivalence classes for the relation where u and v are equivalent if and only if

$$\{z \in B : zu \in L\} = \{z \in B : zv \in L\}.$$

This is the same condition as in the definition of E, except that the order of the concatenations has been reversed.

This form of the Myhill-Nerode theorem is somewhat less interesting than the previous one, since it doesn't lead to the construction of a deterministic finite state automaton in the case where the grammar is regular.

There's a third version of Myhill-Nerode, which does construct a strongly

deterministic finite state automaton in the regular case. This finite state automaton is not minimal in general but it does have one interesting property which the one we constructed earlier lacks. Strong determinism means that if we know the current state and the next input token then we know the next state. This new finite state automaton has the additional property that if we know the current state and the last input token read then we know the previous state.

The third version of Myhill-Nerode is based on continuations and equivalence classes, but instead of consider right continuations, as in the first version, or left continuations, as in the second version, we consider bidirectional continuations.

For any list w we say that (u, z) is a bidirectional continuation of w if $uwz \in L$. We say that w and x are bidirectionally equivalent if they have the same set of bidirectional continuations. Bidirectionally equivalent lists are equivalent in the sense we considered earlier but the converse generally isn't true. The third version of the Myhill-Nerode theorem says that the language is regular if and only if the set of bidirectional equivalence classes is finite.

Bidirectional equivalence has one important property which ordinary equivalence lacks. If u is bidirectionally equivalent to x and v is bidirectionally equivalent to y then uv is bidirectionally equivalent to xy. This allows us to perform the quotient construction on B considered as a monoid with concatenation as the operation. The quotient is called the syntactic monoid of the language. Various properties of the language can be defined in terms of the syntactic monoid. The advantage of doing this is that there is only one syntactic monoid for a language. If we try to define properties of a language in terms of the structure of its grammar we need to show that we get the same result regardless of which grammar is used. Similarly, if we try to define properties of a language in terms of the structure of a finite state automaton then we need to show that we get the same result regardless of which automaton is used. The same problem arises if we try to define properties in terms of regular expressions, but not if we define them in terms of the syntactic monoid.

Context free languages

A context free grammar is a phrase structure grammar where every rule gives a finite set of alternates for a non-terminal symbol, each alternate being a finite list of symbols. The second "finite" is redundant because lists are always finite. It's just there as a reminder.

All of the phrase structure grammars we've considered are of the form described above. They have the property that the possible expansions of a symbol are independent of what symbols appear before or after it. That's where the term "context free" comes from. We could imagine more general grammars where the possible expansions are allowed to depend on other symbols in the list. That would take us into the realm of context sensitive grammars. We won't do that this semester though.

A language is called context free if it can be generated by a context free grammar. Left and right regular grammars are context free grammars so regular languages are context free languages. Not every context free grammar is regular though. We saw a context free grammar for the language of balanced parentheses earlier, so it is context free, but we've already seen that it is not regular.

As another example of a context free language which is not regular, consider the language whose members are strings with some number of x's, followed by the same number of y's, followed by any positive number of z's. We can show that this language is not regular using either the Pumping lemma or the Myhill-Nerode theorem. It is context free though, since we can write down the following simple phrase structure grammar for it.

```
%%
start : xsys zs
      ;

xsys  : | x xsys y
      ;

zs    : z | zs z
      ;
```

Similarly, the language with any positive number of x's, followed by some

number of y's, followed by the same number of z's is context free but not regular. In this case it's not possible to use the usual version of the Pumping lemma but it is possible to use the second version, and it's also possible to use the Myhill-Nerode theorem. It is straightforward to write down a phrase structure for this grammar, very similar to the one above.

Not all languages are context free. This follows from a simple counting argument since the set of phrase structure grammars for a non-empty countable set of tokens is countable but but the set of languages for the same set of tokens is uncountable.

Natural languages tend not to be context free, although some of them aren't far off. Well designed computer languages typically are context free, which makes it relatively straightforward to write parsers for them. Languages designed by people or committees who don't have to implement them often fail to be context free, as do languages where the language specification evolved from a preexisting compiler implementation.

I am cheating slightly though when I claim that well designed languages have context free grammars, because there may be some programs which parse correctly but are not valid due to constraints in the language specification which cannot be implemented in a context free way, like declaration before use requirements. Violating these constraints is not, strictly speaking, a syntax error, but this is admittedly a fine distinction. There are programming languages which are context free in the strictest possible sense but you probably wouldn't enjoy debugging a program written in one.

Closure properties

The union of two context free languages is context free. The construction of a context free grammar for the union from context free grammars for the individual languages is exactly the same as for regular languages. Similarly, reversal, concatenation and Kleene star are okay, with essentially the same constructions already saw for regular languages.

The intersection of two context free languages, or the relative complement of one with respect to another, is generally not context free. For example, we've seen that the language consisting of strings with some number of x's followed by the same number of y's and then any positive number of z's is

context free. We've also seen that the language consisting of strings with any positive number of x's, then some number of y's and then the same number of z's is context free. The intersection of these two languages is the language of strings with some positive number of x's followed by the same number of y's and then the same number of z's. That language is not context free, although we don't yet have the tools to prove this. We will return to this example later, once we have a pumping lemma for context free languages.

Although the intersection of context free languages needn't be regular it is true that the intersection of a regular language and a context free language is a context free language. It is also true that if L is context free and M is regular then $L \setminus M$ is context free, although $M \setminus L$ needn't be.

Pushdown automata

Just as the regular languages are those which can be recognised by a finite state automaton the context free languages are those which can be recognised by what's called a pushdown automaton, essentially a finite state automaton with access to a stack.

In addition to the tokens of the language we allow the finite state automaton to use finitely many additional tokens on its stack.

Earlier we considered a language for zeroeth order logic, which had the tokens p, q, r, s, u, !, $\wedge$, $\vee$, $\neg$, $\supset$, $\overline{\wedge}$, $\veebar$, $\equiv$, $\not\equiv$, $\subset$, (,), [,], { and }. We can construct a pushdown automaton which recognises this language.

Our automaton starts by pushing a p onto the stack. It then reads characters one at a time, processing them as follows. In each case where I've written that the machine pops a character off the stack I mean that it checks whether the stack is empty, fails, and pops the the top character otherwise. "Fail" here and below just means terminates unsuccessfully.

- If there is no character to read then it checks whether the stack is empty and terminates successfully if it is and unsuccessfully if it isn't.
- If the character it read is whitespace it does nothing.
- If the character it read is a p, q, r, s, u it pops a character off the stack. If the character it popped is a p it pushes a ! onto the stack. Otherwise

it fails.

- If the character it read is a ! then it continues on to read the next character.
- If the character it read is $\wedge$, $\vee$, $\supset$, $\overline{\wedge}$, $\veebar$, $\equiv$, $\not\equiv$, or $\subset$ it pops a character off the stack. If the character it popped is a $\wedge$ it continues on to read the next character. If the character it popped is a ! then it pops off another character and if that character is a $\wedge$ it continues on to read the next character. In all other cases it fails.
- If the character it read is a $\neg$ then it pops a character off the stack. If the character it popped is a p then it pops another character off the stack. If that character is a $\wedge$ then it continues on to read the next character. In all other cases it fails.
- If the character it read is a (then it pops a character off the stack. If that character is a p then it pushes a), then a p, then a $\wedge$, and then another p onto the stack and continues on to read the next character. In all other cases it fails.
- If the character it read is a [then it pops a character off the stack. If that character is a p then it pushes a], then a p, then a $\wedge$, and then another p onto the stack and continues on to read the next character. In all other cases it fails.
- If the character it read is a { then it pops a character off the stack. If that character is a p then it pushes a }, then a p, then a $\wedge$, and then another p onto the stack and continues on to read the next character. In all other cases it fails.
- If the character it read is a) then it pops a character off the stack. If the character it popped is a) then it continues on to read the next character. If the character it popped is a ! then it pops another character. If that character is) then it continues on to read the next character. In all other cases it fails.
- If the character it read is a] then it pops a character off the stack. If the character it popped is a] then it and continues on to read the next character. If the character it popped is a ! then it pops another character. If that character is] then it continues on to read the next character. In all other cases it fails.

- If the character it read is a } then it pops a character off the stack. If the character it popped is a } then it and continues on to read the next character. If the character it popped is a ! then it pops another character. If that character is } then it continues on to read the next character. In all other cases it fails.

Before explaining why this works it may be helpful to trace the computational path for a particular input. I'll choose $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ as my input string. As the computation proceeds we need to keep track of what portion of the input has been read and what the contents of the stack is. The accompanying diagram does this.

	$\{$	$[$	$($	p	$\supset$	q	$)$	$\wedge$	$($	q	$\supset$	r	$)$	$]$	$\supset$	$($	p	$\supset$	r	$)$	$\}$
p	p	p	p	!	p	!	$\wedge$	p	p	!	p	!	$]$	$\wedge$	p	p	!	p	!	$\}$	
	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$)$	$)$	p	$]$	$\wedge$	$\wedge$	$)$	$)$	$\wedge$	p	$\}$	$\wedge$	$\wedge$	$)$	$)$		
	p	p	p	p	$\wedge$	$\wedge$	$]$	$\wedge$	p	p	$]$	$]$	p	$\}$		p	p	$\}$	$\}$		
	$\}$	$]$	$)$	$)$	p	p	$\wedge$	p	$)$	$)$	$\wedge$	$\wedge$	$\}$			$)$	$)$				
		$\wedge$	$\wedge$	$\wedge$	$]$	$]$	p	$\}$	$]$	$]$	p	p				$\}$	$\}$				
		p	p	p	$\wedge$	$\wedge$	$\}$		$\wedge$	$\wedge$	$\}$	$\}$									
		$\}$	$]$	$]$	p	p			p	p											
			$\wedge$	$\wedge$	$\}$	$\}$			$\}$	$\}$											
			p	p																	
			$\}$	$\}$																	

The interpretation of the diagram is that the column below each input character is the state of the stack after reading it and doing all the associated stack operations. To the left all the input characters there's a column with just a single p, which represents the state of the stack just before reading the first character.

At no point before the end of the input does the automaton terminate unsuccessfully and the stack is empty at the end so the automaton terminates successfully. In other words, this string is recognised as a member of the language.

You may have guessed how this automaton uses its stack. After processing a character the stack shows one valid continuation for the input at that point. This continuation is chosen in such a way that even if the next input character is not the first character of that continuation, i.e. the character at

the top of the stack, we can easily adjust the stack to get a valid continuation of the new input string, if there is such a valid continuation, and fail if there is none. This is a strategy which happens to work for this language and some others. It does not work for context free languages in general though.

One feature of the pushdown automaton described above is that it always terminates, either successfully or unsuccessfully. This is clear because at each stage we read an input character and eventually we run out of input. Another feature of the automaton is that it is deterministic. Whenever we read an input token, check whether the stack is empty, or pop a token from the stack there is at most one way to continue the calculation, although there may be none in those cases where we terminate unsuccessfully. These two features are desirable but neither of them is required. Pushdown automata are allowed to have multiple options for their next step. As with all non-deterministic computations we consider we say that the input is accepted if there is some computational path which terminates successfully, even if others terminate unsuccessfully or not at all.

Parsing by guessing

The preceding section gave a deterministic pushdown automaton recogniser for a particular grammar. The method used was adapted to that particular language and doesn't provide much inspiration if someone hands us another language and asks for a pushdown automaton recogniser for it. If we're willing to give up determinism then there's a simple recipe we can use:

- Empty the stack, if necessary, and then push the grammar's start symbol onto it.
- Repeat the following indefinitely:
  - If the stack and input are empty then terminate successfully.
  - If the stack is empty and the input is non-empty then terminate unsuccessfully.
  - If neither of these things has happened the stack must be non-empty. Pop the item at the top of of the stack. It will be a symbol from the grammar, either terminal or non-terminal.

- If it's non-terminal then pick one of its alternates from the grammar, terminating unsuccessfully if there are none, and push the symbols from that alternate onto the stack in reverse order.
- If it's a terminal symbol then read an input token, terminating unsuccessfully if the input is empty. Check whether the token belongs to the terminal symbol, terminating unsuccessfully if it does not.

This doesn't have to terminate. What's different about this method from the one we saw in the example is that this one processes one stack item at a time rather than one input token at a time. The stack can shrink or grow, while the remaining input only ever shrinks. In fact it's very unusual for all computational paths to terminate.

It's also non-deterministic because of the step where we choose an alternate from the rule for a non-terminal symbol.

If the input belongs to the language then some computational path will terminate successfully. If the input does not belong to the language then it's possible that all computational paths terminate unsuccessfully, but it's more likely that at least one fails to terminate at all. If so then the automaton will never provide us with an answer because at any finite stage of the computation we won't know whether it would eventually succeed if given more time.

I'll illustrate this method with the same input as above for the language of zeroeth order logic. It will be necessary to write this in a somewhat different format from the previous example because of its size, and because it's organised somewhat differently, processing one stack item at a time rather than one input character at a time.

1. input: $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: statement
2. input: $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: expression
3. input: $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: { expression binop expression }
4. input: $[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: expression binop expression }
5. input: $[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: [expression binop expression] binop expression }
6. input: $(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: expression binop expres-

sion] binop expression }
7. input: $(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: (expression binop expression) binop expression] binop expression }
8. input: $p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: expression binop expression) binop expression] binop expression }
9. input: $p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: variable binop expression) binop expression] binop expression }
10. input: $p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: letter binop expression) binop expression] binop expression }
11. input: $\supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: binop expression) binop expression] binop expression }
12. input: $q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: expression) binop expression] binop expression }
13. input: $q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: variable) binop expression] binop expression }
14. input: $q) \wedge (q \supset r)] \supset (p \supset r)\}$ stack: letter) binop expression] binop expression }
15. input: $) \wedge (q \supset r)] \supset (p \supset r)\}$ stack:) binop expression] binop expression }
16. input: $\wedge(q \supset r)] \supset (p \supset r)\}$ stack: binop expression] binop expression }
17. input: $(q \supset r)] \supset (p \supset r)\}$ stack: expression] binop expression }
18. input: $(q \supset r)] \supset (p \supset r)\}$ stack: (expression binop expression)] binop expression }
19. input: $q \supset r)] \supset (p \supset r)\}$ stack: expression binop expression)] binop expression }
20. input: $(q \supset r)] \supset (p \supset r)\}$ stack: variable binop expression)] binop expression }
21. input: $q \supset r)] \supset (p \supset r)\}$ stack: letter binop expression)] binop expression }
22. input: $q \supset r)] \supset (p \supset r)\}$ stack: letter binop expression)] binop expression }
23. input: $\supset r)] \supset (p \supset r)\}$ stack: binop expression)] binop expression }
24. input: $r)] \supset (p \supset r)\}$ stack: expression)] binop expression }
25. input: $r)] \supset (p \supset r)\}$ stack: variable)] binop expression }
26. input: $r)] \supset (p \supset r)\}$ stack: letter)] binop expression }
27. input: $)] \supset (p \supset r)\}$ stack:)] binop expression }

28. input:] $\supset$ ($p \supset r$)} stack:] binop expression }
29. input: $\supset$ ($p \supset r$)} stack: binop expression }
30. input: ($p \supset r$)} stack: expression }
31. input: ($p \supset r$)} stack: (expression binop expression) }
32. input: $p \supset r$)} stack: expression binop expression) }
33. input: $p \supset r$)} stack: variable binop expression) }
34. input: $p \supset r$)} stack: letter binop expression) }
35. input: $\supset r$)} stack: binop expression) }
36. input: r)} stack: expression) }
37. input: r)} stack: variable) }
38. input: r)} stack: letter) }
39. input:)} stack:) }
40. input: } stack: }
41. input: stack:

At no point before the stack emptied does the automaton terminate unsuccessfully and the input is empty once the stack is so the automaton terminates successfully. In other words, this string is recognised as a member of the language.

Note that this is one possible computational path. It is, in fact, the only one which terminates successfully. There are many others, some of which terminate unsuccessfully and some of which fail to terminate at all. As discussed earlier we could represent the set of all computational paths with a tree, but this tree would be infinite. It is possible to give a deterministic algorithm by, for example, traversing this tree in breadth first order. There's no way for a deterministic pushdown automaton to do this, since maintaining the full tree requires something more powerful than a stack, but it could be done, for example, by a deterministic Turing machine. With a breadth first traversal we would only see a finite portion of the full infinite tree. I have chosen not to illustrate the portion which would be traversed because this planet is unfortunately not large enough to contain it.

Deterministic pushdown automata

We've now seen two pushdown automata for the same language. We saw in the preceding chapter that any language which can be recognised by a finite state automaton can be recognised by a deterministic finite state

automaton. Although we haven't defined Turing machines yet it is true that every language which can be recognised by a Turing machine can be recognised by a deterministic Turing machine. Since pushdown automata are intermediate in power between finite state automata and Turing machines it would seem reasonable to expect that every language which can be recognised by a pushdown automaton can also be recognised by a deterministic pushdown automation. The example considered above lends some support to this expectation, since there's a natural way to recognise the language with a non-deterministic pushdown automaton but with a certain amount of ingenuity one can also construct a deterministic pushdown automaton for the language. Unfortunately though there are context free languages which cannot be recognised by any deterministic pushdown automaton.

From pushdown automata to context free grammars

I've described one way of constructing a pushdown automaton from a context free grammar in such a way that the automaton recognises exactly those lists which are generated by the grammar. The reverse construction is also possible. Given a pushdown automaton we can construct from is a context free grammar which generates those lists which are recognised by the automaton.

As an example, consider a pushdown automaton working with the symbols (and) set up as follows:

- The only state is the start state, so there are no state transitions.
- The stack is initially empty.
- The automaton reads the input one symbol at a time. Whenever it reads a (it pushes it. Whenever it reads a) it attempts to pop whatever is at the top of the stack. If it can't pop an item because the stack is already empty then the computation terminates unsuccessfully.
- If it reaches the end of the output without having failed in the way just described then it terminates, successfully if the stack is empty and unsuccessfully if the stack it not empty.

We want a grammar for the language of lists of symbols which cause this automaton to terminate successfully when given them as input.

The key idea is to introduce a new symbol whose expansion should be precisely those lists of symbols which leave us with an empty stack if we started with an empty stack. In other words, although the automaton might push new symbols onto the stack it will pop all of them off of the stack by the time it reaches the end of the list and won't attempt to pop off more than it has pushed on. Lacking any more creative ideas, I will call this new symbol `new`.

Although I described the set of expansions of `new` in terms of a stack which is empty at the start and end of the list the behaviour of the automaton is essentially the same with any other initial stack. It will end up with the same stack at the end as at the start because it never dips below the current stack of the top. If it did then it would cause a failure when given the empty stack. For this reason we can describe the expansions of `new` as those lists of symbols which, when fed to the automaton at a point where there are k items on the stack will leave the stack with the same k items and no others, with the size of the stack never dipping below k in between.

Although it's obvious, it's worth stating explicitly that the empty list has the property described above and so should be a possible expansion for the symbol `new`.

As described above, the stack has size k at the start and end of any list of symbols which are a valid expansion of `new` and it has size at least k at any point in between. It might or might not have exactly k at some intermediate point.

If it has size k at an intermediate point then that means we can split the list in two at that point and each of those pieces will also be a valid expansion of `new`.

If it doesn't have size k at any intermediate point then it has size greater than k everywhere in between, since the size can never be less than k. In other words right at the start of the list we must have pushed a symbol and this symbol isn't popped off until the end of the list. In between those operations we have a situation where the stack starts with size $k + 1$, ends with size $k + 1$, and has size at least $k + 1$ everywhere in between. In other words, whatever list of symbols we read in between that first push and that last pop must be an expansion of `new`. In this case it's simple to figure out what the first and last things we read are, since only a `(` can cause a push

and only a) can cause a pop. So in the case we're currently considering, the one where the stack doesn't have size k at any intermediate point, we must have a (, then something of type new, and then a).

In the paragraphs above we've seen three things new could expand to: the empty list, a concatenation of two news, or a (followed by a new and a). It's easy to see that these are the only possibilities. So the grammar rule for new must be

```
new : | new new | "(" new ")"
```

What is the start symbol for this grammar? We've specified that the stack is initially empty and the for a successful termination it should be empty finally as well and that no attempt to pop from an empty stack occurs in between. The definition of new fits this description exactly, so we can just take new to be the start symbol and the single rule above is the entire grammar.

As you may already have guessed, the language recognised by this machine is the language of balanced parentheses and the grammar above is a grammar for it. It's not the grammar we used previously. That grammar was unambiguous while this one is ambiguous.

The sort of analysis we employed above can be applied, with some modifications, to any pushdown automaton. The main complication is the possibility of multiple states. If we have multiple states then we need a new symbol for each possible pair of initial and final state for lists of symbols which preserve the stack in the way described above. Only for those where those states are the same is the empty list a valid expansion. When we consider those with an intermediate point where the stack is of the same size as at the beginning and end we have to allow for the various states the automaton could be in at that intermediate point. In the case where this doesn't happen, i.e. where we push something onto the stack at the start and pop something off only at the end, we have to account for all the states we could be in just after pushing and just before popping. We also need to consider all the various choices for a symbol to push and pop. Finally, if we have multiple states then we need to identify the start state or states and the accepting state or states, those which represent a successful termination. The start state will then have an alternate for each pair of start and accepting state. Considering all of these possible combinations of states and symbols leads to complicated grammars, but conceptually the proce-

dure isn't really different from the one we employed in the language of balanced parentheses example.

Pumping lemma

Just as there is a pumping lemma for regular languages there is one for context free languages. The statement is of a similar kind, but more complicated.

For every context free language there is a natural number p such that every w of L of length at least p can be written in the form $w = abcde$ where the lists a, b, c, d and e have the following properties:

- bcd is of length at most p.
- b and d are not both empty.
- For every natural number n the list ab^ncd^ne is a member of L.

Application

Consider the language consisting of strings some positive number of x's, followed by the same number of y's, followed by the same number of z's. We met this language earlier as the intersection of two context free languages. If this language is context free then there is a number p as in the statement of the lemma. Let w be the string with p x's, followed by p y's, followed by p z's. This is a member of the language and is of length at least p so there should be strings a, b, c, d and e satisfying the conditions listed in the statement of the lemma. The substring bcd is of length at most p so it could contain x's or z's but not both. If we take $n = 0$ we get the string ace, i.e. $abcde$ with b and d removed. If bcd had no x's then ace has p x's and is of length less than $3p$. If it bcd had no z's then ace has p z's and is of length less than $3p$. There are no members of the language which satisfy either of those conditions though. This contradicts the statement of the lemma, so our assumption that the language is context free must have been false. In particular, we now have an example of two context free languages whose intersection is not context free.

Proof

By assumption our language is context free so it has a phrase structure grammar of the type described previously. Let s be the number of and r be the maximum number of symbols appearing in any rule.

In any parse tree for a member of the language the number of leaves among the descendents of a node is at most r^h, where h is the maximum of the lengths of the branches starting from that node. The length of branches here is the number of edges in a path from the node to the leaf, which is one less than the number of nodes along that branch.

Let $p = r^{s+1}$ and let w be a member of L of length at least p. The parse tree for w must then have a branch of length at least s from the root node. I've written "the" parse tree but there could be more than one if the grammar is ambiguous. What the argument above really shows is that every parse tree for w has such a branch. If there's more than one parse tree we'll choose one with as few nodes in its parse tree as possible.

On the parse tree for w choose a branch of maximal length. From what we said above this length is at least $s + 1$. The number of symbols appearing is on this branch is one more than then length and so is greater than $s + 1$. If we look at the last $s + 1$ symbols then one must be repeated. Any such symbol in non-terminal because terminal symbols don't have children in the parse tree. Choose one, and choose two occurrences of it. We'll call the one closer to the root the outer occurrence and the one farther from the root the inner occurrence. Let c be the expansion of the inner occurrence and let f be the expansion of the outer occurrence. Let a be the part of w before f and let e be the part after w, so that $w = afe$. Let b be the part of f before c and let d be the part after c, so that $f = bcd$.

Now $w = abcde$. f, i.e. bcd, is of length less than p because the maximal branch length from the outer occurrence is at most $s + 1$. Taking the parse tree for w and replacing the part of the tree descending from the outer occurrence with the part of the tree descending from the inner occurrence has the effect of replacing f by c in afe and so gives a parse tree for ace, which must therefore also be a member of the language. This parse tree has fewer nodes than the minimal parse tree for w so ace is not w. In other words, b and d are not both empty. Also, ab^0cd^0e is a member of the language. We could also replace the part of the tree descending from the inner occurrence

with the part descending from the outer occurrence, to get a parse tree for $abfde$, for ab^2cd^2e, which must therefore also be a member of the language. This construction is repeatable, so we can replate that c by an f to get ab^3cd^3e and so on. In this way we see that ab^ncd^ne is a member of the language for all natural numbers n.

Other idealised machines

More finite state automata

Our finite state automata can be thought of as machines with a restricted stack, one we are only allowed to pop symbols off of. Initially this stack contains the input, with the first symbol at the top of the stack and the last symbol at the bottom.

We could, if we wanted to, consider finite state automata which have a write-only stack rather than a read-only stack, i.e. ones which we can push symbols onto but can pop symbols off of. These have no input but have an output, i.e the final state of the stack, where can conventionally agree that the top of the stack is the last output symbol and the bottom is the first.

It's reasonable to ask which languages can be the set of outputs for such a finite state automaton. If we restrict to deterministic finite state automata then the answer is totally uninteresting, since these could only ever generate one list. If we allow non-deterministic finite state automata then the answer is once again the regular languages, so we have yet another characterisation of these languages.

We could even consider a generalisation of finite state automata with two stacks, one read-only stack for the input and one write-only stack for the output. These are generally called transducers. The other types of finite state automata can be considered as special cases of the transducer. You can, for example, consider the read-only automata from the earlier chapter as ones which read the input and produce as output a single special symbol, either `accept` or `reject`, depending on whether we end up in an accepting or rejecting state. It's easy to generalise this to a classifier, a machine which produces output from a finite set of options, not just two. In fact this essentially is the job of a lexical analyser and a lexical analyser can

be implemented by such an automaton, a special case of the transducer.

More pushdown automata

We can similarly consider the input to a pushdown automaton as as a separate stack which, unlike its working stack, is read-only, i.e. allows only pop operations and not push operations.

Every finite state automaton is a pushdown automaton, specifically a pushdown automaton which doesn't ever touch its working stack, so anything we can do with a finite state automaton can be done with a pushdown automaton. The converse is not true. We've already seen that every context free language is recognised by a pushdown automaton and that every finite state automaton recognises a regular language. If every pushdown automaton could be simulated with a finite state automaton then every context free language would be regular, but we've already seen that this is not true. The language of balanced parentheses, for example, is context free but not regular.

As with finite state automata we could consider instead a version where the other stack is write-only, so the automaton has no input but has an output. Again this is uninteresting if we restrict ourselves to deterministic automata but if we allow non-deterministic automata we get something useful. In fact the languages which can be sets of outputs of such a machine is exactly the context free languages. Arguably this is a more intuitive way to think of the connection between languages and grammars since our generative grammar is used to construct an actually generator rather than a recogniser.

Turing machines

So far we've considered idealised machines with up to two stacks and at least one stack has always been restricted to either only push operations or pop operations. What if we remove some of these restrictions.

The next simplest machine we could consider is one with two stacks whose use is unrestricted. One will be the input stack and one will be the output stack, with the output stack initially empty and the input stack finally empty, but in between we are allowed to push symbols onto the input stack

or pop symbols off of the output stack if we wish. I'll call such a machine a Turing machine, although this terminology requires some comments.

Just as every finite state automaton was a pushdown automaton, every pushdown automaton is a Turing machine, so anything which can be done be a pushdown automaton can be done by a Turing machine.

The definition I've just given for a Turing machine differs from the standard one in two respects. First, I never specified that the machine is deterministic. The standard definition requires this. Second, I allowed the machine to operate independently on its two stacks. The standard definition of a Turing machine uses a single tape rather than a pair of stacks. At each point the machine is at a given position on the tape, from which it's allowed to move left or right. It's allowed to read or change symbols only at this position. We can simulate this with a pair of stacks, one for the symbols to the left of this position and one for the symbols to its right. Any tape operation then corresponds to coordinated operations on the stack. Moving to an adjacent position, for example, involves popping from one stack and pushing to the other. So the standard Turing machine is restricted in two senses compared to what I've defined above. It's deterministic and its permitted stack operations are only those combinations which correspond to tape operations. It turns out that both of these restrictions are more apparent than real though. We saw earlier that any finite state automaton can be simulated by a deterministic finite state automaton, via the power set construction. The corresponding statement for pushdown automata is false but the statement for Turing machines is true. It's also possible simulate a machine with unrestricted stack operations with one where the stack operations are paired in the way they would be for a standard Turing machine with a tape. So although my definition of a Turing machine isn't identical to the standard one the set of computations the two types of machines can do are identical.

A Turing machine

As proved in the last chapter, the language consisting of strings with a positive number of x's followed by the same number of y's followed by the same number of zs is not context free and so cannot be recognised by a pushdown automaton. We can construct a Turing machine which recog-

nises it though.

There are two stacks, one of which initially holds the input and the other of which is initially empty. I'll refer to these as Stack I and Stack E, respectively. While only `x`'s `y`'s and `z`s appear on the stacks initially it's convenient to allow the machine to push and pop `X`'s, `Y`'s and `Z`'s as well.

There are two phases to the computation. In the first phase we perform the following action repeatedly:

- If Stack E is not empty we pop symbols off of it and push them onto Stack I until Stack E is empty.
- We pop symbols off of Stack I and push them onto Stack E until the symbol we pop off of Stack I is an `x`. Once this happens we push an `X` onto Stack E instead of an `x` and move on to the next action. If Stack I becomes empty before we see an `x` then we move on to the second stage of the computation.
- We pop symbols off of Stack I and push them onto Stack E until the symbol we pop off of Stack I is a `y`. Once this happens we push a `Y` onto Stack E instead of a `y` and move on to the next action. If Stack I becomes empty before we see a `y` then we halt and the computation is unsuccessful.
- We pop symbols off of Stack I and push them onto Stack E until the symbol we pop off of Stack I is a `z`. Once this happens we push a `Z` onto Stack E instead of a `z` and move on to the next action. If Stack I becomes empty before we see a `z` then we halt and the computation is unsuccessful.

The second stage is simpler.

- If Stack E is not empty we pop symbols off of it and push them onto Stack I until Stack E is empty.
- We pop symbols off of Stack I one at a time and push them onto Stack E. If any of these symbols is a `y` or a `z` then we halt and the computation is unsuccessful.
- If Stack I becomes empty without any `y`'s or `z`'s having been seen then we halt and the computation is successful.

Conceptually this is fairly simple. In the first stage we replace the first x by an X, the first y by a Y and the first z by a Z, then the second x by an X, the second y by a Y and the second z by a Z, etc. If the input was in the languages then at this point we should have with a positive number of X's followed by the same number of Y's followed by the same number of Zs. If it's not in the language then either we have too few y's or z's in the input or too many. If we have too few then the machine will already have halted unsuccessfully when it empties Stack I while scanning for a y or a z. If we have too many then there will be some y's which didn't get changed to Y's or some z's which didn't get changed to Z's. In that case the machine will halt unsuccessfully during the second stage. Otherwise it will halt successfully after the second stage.

This Turing machine has the rather nice property that no matter what input we give it it will eventually halt, either successfully or unsuccessfully. This is not true of Turing machines in general. A recogniser is required to halt successfully on any input in the language and only on inputs in the language but it is allowed to halt unsuccessfully or run forever on inputs which are not in the language. A Turing machine which always halts unsuccessfully on inputs which are not in the language, like this one does, is called a decider for the language. Every decider is a recogniser, but not vice versa. In fact there are languages for which it's possible to construct a recogniser but impossible to construct a decider.

The Church-Turing hypothesis

As discussed discussed previously, some additional restrictions we could place on Turing machines, like determinism or coordinating operations on the two stacks, turn out not to matter, as long as we're only concerned with what can or can't be done by such a machine and not the precise mechanics of how it is done. What about removing restrictions rather than adding them? Could we, for example, do things with a three stack machine which we can't do with a two stack machine? The answer, both for three and for any higher number, turns out to be no. It's possible to simulate a machine with more stacks on one with just two, just as we could simulate a non-deterministic machine on a deterministic one.

The example above turns out to be typical. Every less restrictive idealised

machine people have been able to imagine, as long as we allow only operations which could be mechanically realised in finite time, turns out to be no more powerful than a Turing machine in terms of what it can compute. This observation has led to what's called the Church-Turing thesis, which is essentially the statement that computable means computable by a Turing machine, i.e. that there is no more powerful notion of computability waiting to be discovered. This thesis isn't susceptible to rigorous proof, or even really a rigorous formulation, but it is almost universally believed by people who study the theory of computation.

Universal Turing machines

If you've ever used a computer you may find the sorts of idealised machines we've been considering odd. We have a separate machine for each computational task. We can, for example, "build" a pushdown automaton recogniser for any context free language but what we'd obviously prefer is a machine which takes a grammar and a list of tokens and tells you whether the list belongs to the language, so that we can use the same machine on any language.

Turing's work predates physical computers so at the time he was writing it really was necessary to have separate machines for each computational task. Turing was aware that this was undesirable though, and so also considered the notion of a universal machine. He had in some sense been anticipated in this regard by Babbage and Lovelace. In Turing's formulation the universal machine was a Turing machine which took as input a description of a Turing machine and of an input to that machine and told you what the output of that machine would be on that input. He described, in some detail, how to construct such a machine. In this picture there are two Turing machines, the simulator and the simulated. The simulator corresponds roughly to modern computer hardware and the simulated to modern computer software. Because the simulator is universal you only need one computer, not a separate one for each computational task.

The Halting Problem

Turing machines don't have to halt. They can run forever. It would be nice if we could determine in advance whether a given machine will halt

on a given input, so we don't waste time waiting for an answer will never come. Note that although a universal Turing machine can simulate any Turing machine on any input we can't use it for this purpose. If we give a universal Turing machine as input a description of a Turing machine and an input on which that Turing machine doesn't halt then the universal Turing machine also won't halt. What we want is a Turing machine which will take as input a description of a Turing machine and an input for that machine and will always terminate, successfully if that Turing machine would halt on that input and unsuccessfully if it would not. This is called the Halting Problem.

There is, unfortunately no solution to the Halting Problem. This can be proved as follows. Assume there is a Turing machine which solves the Halting Problem. There will be several Turing machines to consider in this proof, so we'll number them. The one just considered will be Machine 0. Machine 1 is some Turing machine which takes a pair of inputs of some kind and which always halts, either successfully or not, no matter what those inputs are. Machine two takes a single input and simulates running Machine 1 on two copies of that input. It halts unsuccessfully if Machine 1 would have halted unsuccessfully and does not halt at all if Machine 1 would have halted successfully. Machine 0 and Machine 1 cannot be the same. To see this, give them both, as input, two copies of the description of Machine 2. If Machine 1 would terminate unsuccessfully on this input then Machine 2 would halt unsuccessfully when given itself as input which means Machine 0 would halt successfully. If Machine 1 would terminate successfully then Machine 2 wouldn't halt and so Machine 0 would halt unsuccessfully. In either case Machine 0 and Machine 1 have different behaviour on this input and so they are not the same machine. But Machine 1 was an entirely arbitrary Turing machine except for the requirement that it always halts eventually and if the Halting problem has a solution then Machine 0 is a Truing machine of that type, so we have a contradiction.

You may notice a similarity between this argument and Cantor's diagonalisation argument from set theory, and also with the rather vague explanation I gave to make Tarski's Theorem plausible. Indeed all of these arguments are related. Cantor's was the first and inspired all of the others.

The unsolvability of the Halting Problem shows that some well defined computational problems are provably unsolvable. Perhaps more surpris-

ingly it also provides an effective method for proving large classes of problems to be unsolvable. The most commonly used technique for proving unsolvability is to assume that there is a Turing machine which solves the problem and then show how to construct from it a machine which solves the Halting Problem.

Conclusion

An alternate title for these notes could have been Foundations of Mathematics, but mathematics doesn't really have a single foundational theory. We need a mix of logic, formal linguistics and computability. None of these can really be understood without the others. Logic has to be expressed in a formal language and the question of whether we can test the validity of statements in this language is one of computability. Formal linguistics is useless if languages can't be parsed and presents difficulties of interpretation if they are ambiguous. These are computational questions and their analysis requires some form of logical analysis. Similarly the theory of computation is, as we just saw, closely intertwined with linguistic and logical questions.

It's not just that logic, languages and computability are each dependent on the others. To some extent they are the same subject. The theory of finite state automata, for example, isn't just related to the theory of regular languages; they are the same theory.

In addition to the complications just discussed, there is not not just one version of most of these theories. Different authors use different definitions. Some of these differences are merely annoying, like the different definitions of "countable" for sets, but don't change the content of the theories. Some differences reflect the tension between making it easy to reason within the system and making it easy to reason about the system. Some, like the various axiom schemes for set theory, result from fundamental disagreements about what is true. There are more of these disagreements than I've had space to address in these notes. For example, even in zeroeth order logic there are adherents of classical logic, which is the version I've presented here, but also of intuitionist logic and minimal logic.

The main purpose of studying the foundations of mathematics is therefore

not to find an agreed formal system which we can all use to make sure that everything is guaranteed to be correct. The main thing we gain from studying these formal systems is a better understanding of informal reasoning, which is what mathematicians spend almost all of our time doing. If we can, for example, set out precise rules for substitution then we can identify and avoid improper substitutions in informal arguments, and also be more confident in correct uses. Similarly, an understanding of formal languages is useful in understanding and creating mathematical notation. Informal mathematical language is rarely unambiguous, but it's important to understand the ambiguities which do exist and to try not to create more of them.