MAU22C00 lecture notes

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 logical 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? 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 example, 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 pro-

cedure 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 logical operator "and", "or" and "not" according to fixed rules.

We'll avoid the ambiguity of whether "Probability and Statistics" is one module or two by insisting that every module title is one word, beginning with a capital letter. In a real system you would probably use some other mechanism, like using module codes instead of names, or using symbols unlikely to occur in a module name to stand in for "and", "or" and "not".

We'll resolve the ambiguity about how to split up a compound phrase like "Probability and Statistics or Algebra and Geometry" by declaring that "not" takes precedence over "and", which in turn takes precedence over "or". By precedence we mean that it binds more tightly, so given the choice between binding the names "Probability" and "Statistics" with an "and" or "Statistics" and "Algebra" with an "or" we prefer to bind "Probability" and "Statistics" together first. Only after "Probability" and "Statistics" have been bound together with "and" and "Algebra" and "Geometry" do we bind the two larger phrases "Probability and Statistics" and "Algebra and Geometry" together with an "or". While not strictly necessary, it is convenient to allow the use of parentheses to override these precedence rules. The alternative interpretation described earlier could then be written as "Probability and (Statistics or Algebra) and Geometry". This could

also expressed without parentheses as "Probability and Statistics and Geometry or Probability and Algebra and 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.

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 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, together 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 variables and 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 and 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 and Statistics and not Algebra and not Geometry and not ..." 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 and Statistics and, for all x, x equals Probability or x equals Statistics or not x". The "for all" is a quantifier, the universal quantifier, and x 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 what's called an abstract syntax tree.

The following three figures give a visual representation of the abstract syntax trees for the two statements considered above.

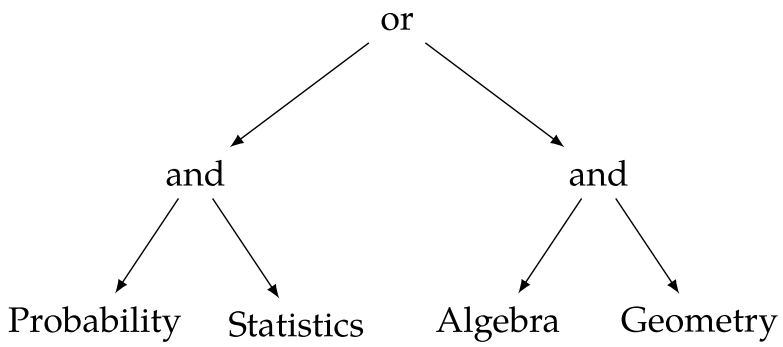


Figure 1: Syntax tree for ((Probability and Statistics) or (Algebra and Geometry))

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 our

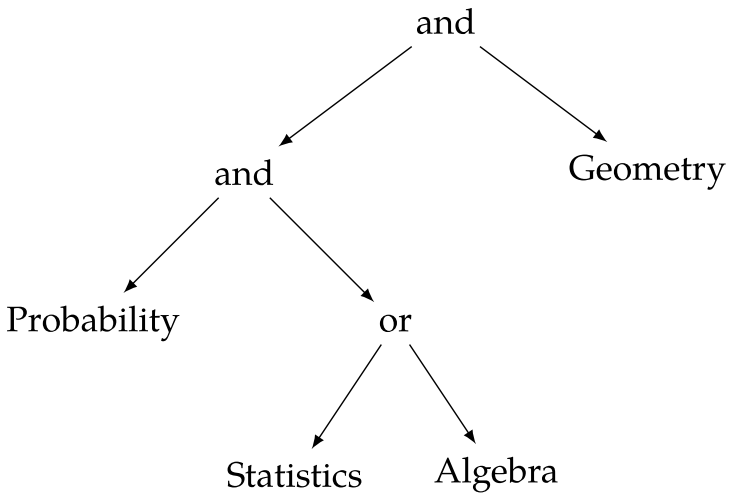


Figure 2: Syntax tree for ((Probability and (Statistics or Algebra)) and Geometry)

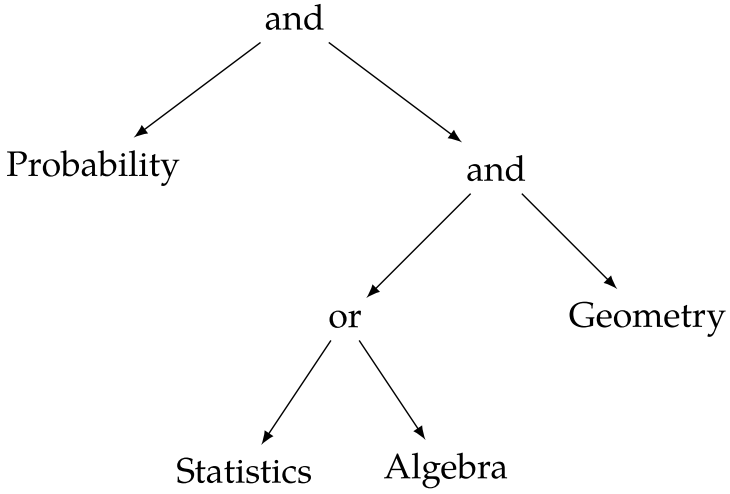


Figure 3: Syntax tree for (Probability and ((Statistics or Algebra) and Geometry))

case the leaf nodes are all labelled by module names and the other nodes are all labelled by logical operators.

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 these abstract syntax trees are as follows.

```
((Probability and Statistics) or (Algebra and Geometry))

((Probability and (Statistics or Algebra)) and Geometry)

(Probability and ((Statistics or Algebra) and Geometry))
```

The internal representation a computer would use for a tree data structure isn't either 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 and (Statistics or Algebra) and Geometry", depending on which "and" has higher precedence, shows that our grammar isn't fully unambiguous.

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. We'll discuss graphs more later.

Interpretation

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". 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 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 "and" and "not" mean what you expect them to mean and "or" is always to be interpreted inclusively. With this interpretation the remaining ambiguity we saw earlier, concerning precedence between "and"'s or between "or"'s is seen to be harmless, because "and" and "or" 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.

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.

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 and Statistics and not Algebra and not Geometry or not Probability and not Statistics and Algebra and Geometry" in order to achieve the desired interpretation.

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 logical operator "not" 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 "xor", 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 statement "Statistics implies 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 "implies" is "not Statistics or Probability". If this looks wrong then you may need to remind yourself of our precedence rules. Since "not" is higher precedence than "or" the statement will be parsed as "(not Statistics) or Probability" rather than "not (Statistics or 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)and or(Geometry not" 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 logical operators "and" and "or" 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 and Statistics or Algebra and Geometry" is

```
(or (and Probability Statistics) (and Algebra Geometry))
```

while the postfix version is

```
((Probability Statistics and) (Algebra Geometry and) or)
```

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 logical connectives "and", "or" and "not" and the module names. We'll call these tokens. With my rather drastic requirement that module names are single words starting with capital letters this part is easy, but for many interesting languages it is more difficult. 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 "and" or an "or" it increments the counter. When it reads a module name it decrements the counter. When it reads a "not" it does nothing. If the value of the counter reaches 0 at the end, but not before, then the input is grammatically correct. Otherwise it isn't. Here's the input "or and Probability Statistics and Algebra Geometry" together with the value of the counter at each point in the input:

```
1 or 2 and 3 Probability 2 Statistics 1 and 2 Algebra 1 Geometry 0
```

We can turn this into a 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 or 2 and 3 Probability 2 Statistics 1 and 2 Algebra 1 Geometry 0
```

```
2 and 3 Probability 2 Statistics 1
1 and 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. Depending on our conventions we can label the node with the whole phrase or just the first token. The version where we give just the first token is one of the abstract syntax tree diagrams we saw earlier. The version with the whole phrase is often called the "parse tree", although some authors use those terms interchangeably.

A parser for the postfix language

It's equally easy to write a parser for the postfix version. Again we'll start with a checker. The checker has a counter initialised to 0. When it reads a module name it increments the counter. When it reads an "and" or an "or" it decrements the counter. When it reads a "not" 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 and Algebra Geometry and or" decorated with the values of the counter at each stage:

```
0 Probability 1 Statistics 2 and 1 Algebra 2 Geometry 3 and 2 or 1
```

Constructing an abstract syntax tree from the checker 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, assuming we're using the version where each node is labelled with a single token rather than the whole phrase.

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 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. 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 "not" 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 "and" 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 "or" 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 "not" it pops a value from the top of the stack and pushes 1 minus that value onto the stack. When it reads an "and" it pops two values off of the stack and pushes their maximum onto the stack. When it reads an "or" 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 and Algebra Geometry and or" 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 and 1 Algebra 0 1 Geometry 1 0 1 and 1 1 or 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 grammar checker 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.

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 excepted 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

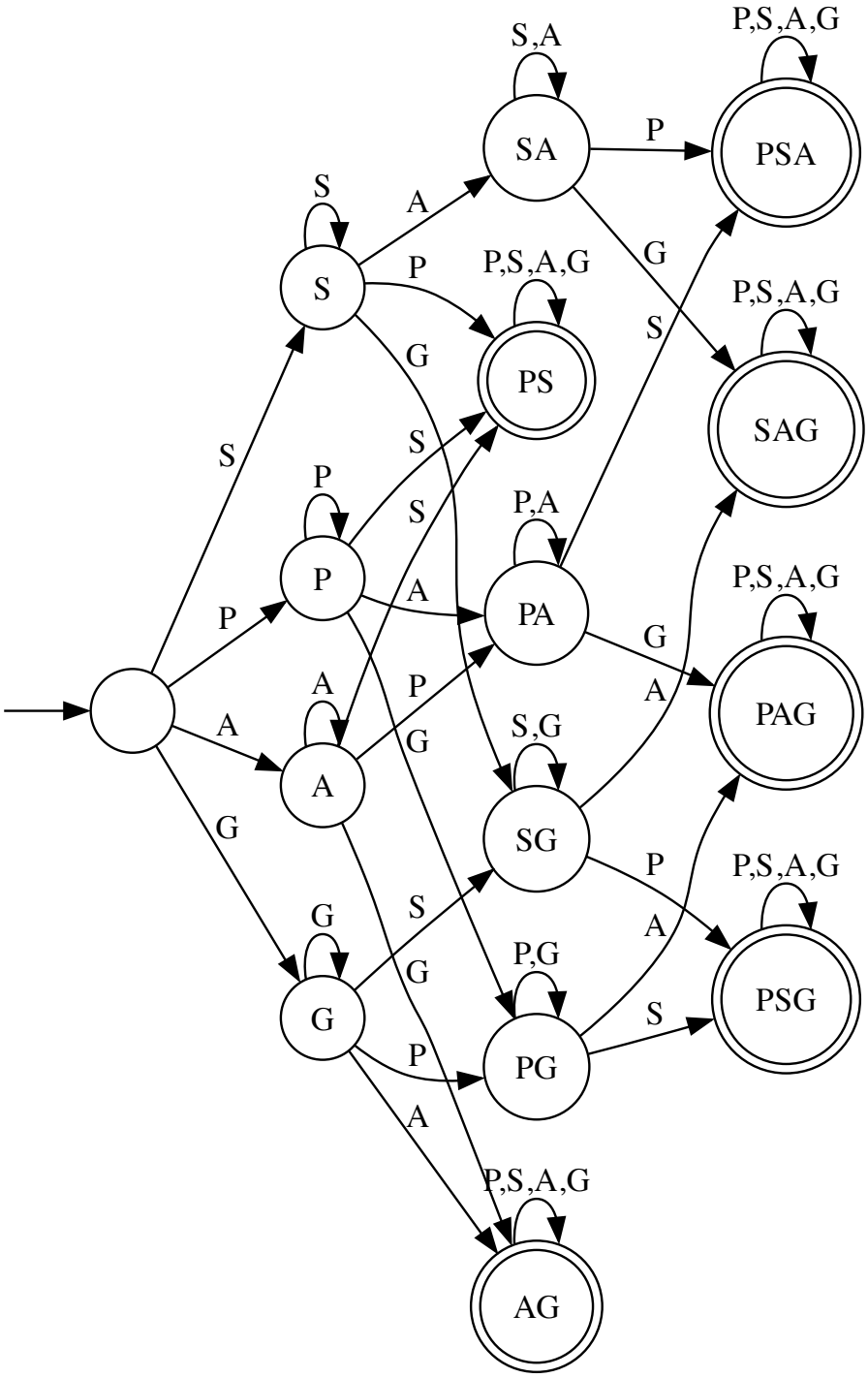


Figure 4: A finite state automaton

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.

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 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.

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 or not 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 can always be 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.

Rules of inference

The definition for a consequence given above requires checking a very large number of possibilities. To verify that "Probability and Statistics or Algebra and Geometry" is a consequence of "Probability and 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 or 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 "or" 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 and B" is a grammatically correct statement and that A and B are both consequences of it. This unfortunately isn't true. "Probability and Statistics or Algebra and Geometry" is grammatically correct statement, as are "Probability" and "Statistics or Algebra and Geometry". The second of these is indeed a consequence of "Probability and Statistics or Algebra and Geometry" but the first is not. There are module selections for which the statement "Probability and Statistics or Algebra and 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 "and" is an unnatural place to break the expression "Probability and Statistics or Algebra and Geometry". It's possible to express this in terms of the abstract syntax tree. Breaking a statement into two pieces using an "and" at the root of its abstract syntax tree is safe. Breaking it at an "and" 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 "and 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 and". For the fully parenthesised infix language it's true that if A and B are grammatically correct then so is "(A and 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. 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 and Statistics or Algebra and 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 logical operators "and", "or" and "not" so it's easy to write. It's helpful if it has an option to throw an error whenever it sees a logical 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

A grammar example (bc)

`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. I'll start by giving the grammar, then introduce some terminology useful for talking about it, then point out some features. Later I'll describe in more detail the grammar of the language in which the grammar is expressed.

The grammar of the `bc` calculator, as given in the specification, is as follows. Understanding it in detail is not necessary unless for some reason you want to write a POSIX-compliant implementation of the `bc` utility.

```
%token    EOF NEWLINE STRING LETTER NUMBER


%token    MUL_OP
/*        '*', '/', '%'                           */


%token    ASSIGN_OP
/*        '=', '+=', '-=', '*=', '/=', '%=', '^=' */


%token    REL_OP
/*        '==', '<=', '>=', '!=', '<', '>'        */


%token    INCR_DECR
/*        '++', '--'                              */


%token    Define    Break    Quit    Length
/*        'define', 'break', 'quit', 'length'     */
```

```
%token    Return    For    If    While    Sqrt
/*        'return', 'for', 'if', 'while', 'sqrt'  */

%token    Scale    Ibase    Obase    Auto
/*        'scale', 'ibase', 'obase', 'auto'       */

%start    program

%%

program             : EOF
                    | input_item program
                    ;

input_item          : semicolon_list NEWLINE
                    | function
                    ;

semicolon_list      : /* empty */
                    | statement
                    | semicolon_list ';' statement
                    | semicolon_list ';'
                    ;

statement_list      : /* empty */
                    | statement
                    | statement_list NEWLINE
                    | statement_list NEWLINE statement
                    | statement_list ';'
                    | statement_list ';' statement
                    ;
```

```
statement            : expression
                     | STRING
                     | Break
                     | Quit
                     | Return
                     | Return '(' return_expression ')'
                     | For '(' expression ';'
                          relational_expression ';'
                          expression ')' statement
                   | If '(' relational_expression ')' statement
                  | While '(' relational_expression ')' statement
                     | '{' statement_list '}'
                     ;


function             : Define LETTER '(' opt_parameter_list ')'
                          '{' NEWLINE opt_auto_define_list
                          statement_list '}'
                     ;


opt_parameter_list   : /* empty */
                     | parameter_list
                     ;


parameter_list       : LETTER
                     | define_list ',' LETTER
                     ;


opt_auto_define_list : /* empty */
                     | Auto define_list NEWLINE
                     | Auto define_list ';'
                     ;
```

```
define_list          : LETTER
                     | LETTER '[' ']'
                     | define_list ',' LETTER
                     | define_list ',' LETTER '[' ']'
                     ;


opt_argument_list    : /* empty */
                     | argument_list
                     ;


argument_list        : expression
                     | LETTER '[' ']' ',' argument_list
                     ;


relational_expression : expression
                     | expression REL_OP expression
                     ;


return_expression    : /* empty */
                     | expression
                     ;


expression           : named_expression
                     | NUMBER
                     | '(' expression ')'
                     | LETTER '(' opt_argument_list ')'
                     | '-' expression
                     | expression '+' expression
                     | expression '-' expression
                     | expression MUL_OP expression
                     | expression '^' expression
                     | INCR_DECR named_expression
```

```
                    | named_expression INCR_DECR
                    | named_expression ASSIGN_OP expression
                    | Length '(' expression ')'
                    | Sqrt '(' expression ')'
                    | Scale '(' expression ')'
                    ;


named_expression    : LETTER
                    | LETTER '[' expression ']'
                    | Scale
                    | Ibase
                    | Obase
                    ;
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
        ;
```

This isn't actually a complete grammar. There are some further rules which are given afterwards in ordinary text, but those are mostly boring bits like the fact that `NEWLINE` is the newline character.

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. If they're not single characters then a lexical analyser is needed to group characters into tokens. The grammar above starts with some information about the tokens in the the language `bc` accepts. `=` and `+=` are both tokens, for example. Each token is assigned to one, and only one, group, called its symbol. In the example `=` and `+=` are in the group `ASSIGN_OP`. Assigning tokens to symbols part of the lexical analyser's job. While we may allow infinitely many tokens there should only be finitely many symbols. In the `bc` grammar `STRING` is a symbol with infinitely many tokens. Although it's not specified in the formal part of the grammar the lexical analyser recognises almost any string of characters as a `STRING`. Some tokens are likely to be in a group by themselves, in which case people tend to blur the token/symbol distinction. Technically, for example, there are two `NEWLINE`s, the token and the symbol, a set of tokens whose only element is the `NEWLINE` token. A lot of people blur the distinction even when there are multiple symbols in a group and use the words symbol and token as interchangeable.

The symbols we've just discussed, the ones which are groups of tokens are called terminal symbols or just terminals. There are other symbols, conveniently called nonterminal symbols or just nonterminals. In the `bc` example `program`, `input_item` and `semicolon_list` are nonterminals. In fact the non-terminals precisely the things you see listed on the left hand sides of all the grammar rules after the line

```
%%
```

One of these symbols has a special status. It is called the start symbol. In the example above `program` is the the start symbol. You can tell because of the line

```
%start    program
```

Not everyone labels the start symbol. If it's not labelled then the convention is that it's the one on the left hand side of the first grammar rule. 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 a `program` can be an `EOF` symbol, which is just the end of file marker, or an `input_item` followed by a `pro-`

`gram`. The `EOF` symbol is terminal while `input_item` and `program` are non-terminal. An `input_item` can be a `semicolon_list` followed by a `NEWLINE` or a `function`. `NEWLINE` is a terminal while `semicolon_list` and `function` are non-terminal. The | character separates the distinct possibilities in each case.

You can see from the rule for `program` 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.

As might be expected from a field where people from radically different fields, like computer science, linguistics and mathematics, not everyone uses the same terminology and the specifics of how grammar rules are written can vary a lot. Most differences are minor but one is quite significant: whether the "other symbols" referred to above, used to build up a nonterminal symbol, could include no symbols. The three conventions in common use are to allow this in all cases, to allow it only for the start symbol, and do allow it for no symbols. The authors of the specification belong to the tradition in which this is allowed for all symbols, as you can see from the rules for `semicolon_list`, `statement_list`, `opt_parameter_list` and a number of others. The `/* empty */` things you see there, like everything between a `/*` and a following `*/`, are comments, which are there to draw attention to the fact that each of these symbols could be built from no symbols, and the meaning would be the same if they were omitted.

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 nonterminals. 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.

A subexample

The full language for `bc` is rather complicated so let's concentrate just on the last bit for now:

```
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
        ;
```

You shouldn't assume just because a name is familiar that it means what

you think. According to this grammar `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 a further part of its grammar is a rule for `expression` which includes `'-' expression`. A `NUMBER` is an `expression` and `-` followed by any `expression` is an `expression` so `-7` isn't a `NUMBER` but it is an `expression`.

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 `NUMBER`s. 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.

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. 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 the 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

The grammar for `integers` considered above is unambiguous, in the sense that there's only one abstract syntax tree we can get from any given input. The same is true of `NUMBERS` in `bc`. The `bc` grammar as a whole is ambiguous though. One of the possible expansions is `expression + expression`. There are therefore at least two ways to generate the expression `1 + 2 + 3`. One is

```
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 `+`.

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 that this particular grammar be used, 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.

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 a 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. There's also a termination condition and a criterion for success. If we have just a list of tokens then there's nothing further to do and either the list matches the input, in which case the path by which we arrived at this point has all the information necessary to construct an abstract syntax tree, or it doesn't match the input, in which case we can't construct a syntax tree from this particular path, but might have been able to with a different set of choices. It's the choices which make the computation nondeterministic, choices of which symbol in the current list to process, which of the possible right hand sides to expand it with if

it's nonterminal, and which token to choose if more than one corresponds to the same terminal symbol.

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 route 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 A^* 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 A^* 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, E, LISP, 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 lower levels, 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.

A good rule of thumb when developing a language for a specific purpose is to choose one as low as possible in the hierarchy, and to describe it by a grammar at that level, or not much higher. Most modern programming languages are technically context sensitive, but try to segregate their context sensitive features as much as possible. The remainder is context free, with significant parts which are regular or even finite.

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 that of the POSIX standard can be as simple as

```
%token MODULE
/* any possible module name */

%token And Or Not
/* 'and', 'or', 'not' */

%start statement

%%

statement  : statement2
           | statement Or statement2
           ;

statement2 : statement3
           | statement2 And statement3
           ;
```

```
statement3 : statement4
           | Not statement4
           ;

statement4 : MODULE
           | ( statement )
           ;
```

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, "not Algebra and Geometry" will be parsed as if it were "((not Algebra) and Geometry" rather than as "(not (Algebra and Geometry))" because "not" can only appear before a level 4 statement. "Algebra", as a module name, is a level 4 statement but "Algebra and 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

```
%start statement

%%

statement  : statement2
           | statement spaces Or spaces statement2
           ;

statement2 : statement3
           | statement2 spaces And spaces statement3
           ;

statement3 : statement4
           | Not spaces statement4
           ;

statement4 : module
           | '(' statement ')'
```

```
          | '(' spaces statement ')'
          | '(' statement spaces ')'
          | '(' spaces statement spaces ')'
          ;

spaces    : ' '
          | spaces ' '
          ;

module    : capital
          | capital letters
          ;

letters   : letter
          | letters letter
          ;

capital   : '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'
          ;

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'

And       : 'a' 'n' 'd'
          ;

Or        : 'o' 'r'
          ;

Not       : 'n' 'o' 't'
          ;
```

Lexical analysers use spaces or other whitespace to separate tokens but don't typically pass this whitespace on to the parser so if we're dispensing with the parser then we need to handle this in the parser.

A final example

I haven't described the grammar for our grammar specification language. You've probably picked up on most of it from the examples but just in case you haven't here are the details. `%token` statements list the types of terminals the parser can expect from the lexical analyser. Tokens which are single character don't need to be listed. The `%start` statement identifies the start symbol. `%%` separates these from the actual grammar rules. Each grammar rule has the nonterminal being expanded, followed by a colon, followed by the possible expansions, followed by a semicolon. Different possible expansions are separated by `|` characters. Each expansion is just a list of symbols.

This is one of many conventions for listing grammar rules. The authors of the POSIX specification chose it because it happens to be the format expected by the parser generator `yacc`, which, like `bc`, is one of the utilities described in the specification. So its grammar is also defined by the standard. If you're curious here it is:

```
/* Grammar for the input to yacc. */
/* Basic entries. */
/* The following are recognized by the lexical analyzer. */


%token    IDENTIFIER     /* Includes identifiers and literals */
%token    C_IDENTIFIER    /* identifier (but not literal)
                             followed by a :. */
%token    NUMBER          /* [0-9][0-9]* */


/* Reserved words : %type=>TYPE %left=>LEFT, and so on */


%token    LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION


%token    MARK            /* The %% mark. */
%token    LCURL           /* The %{ mark. */
%token    RCURL           /* The %} mark. */
```

```
/* 8-bit character literals stand for themselves; */
/* tokens have to be defined for multi-byte characters. */


%start    spec


%%


spec  : defs MARK rules tail
      ;
tail  : MARK
      {
        /* In this action, set up the rest of the file. */
      }
      | /* Empty; the second MARK is optional. */
      ;
defs  : /* Empty. */
      |    defs def
      ;
def   : START IDENTIFIER
      |    UNION
      {
        /* Copy union definition to output. */
      }
      |    LCURL
      {
        /* Copy C code to output file. */
      }
        RCURL
      |    rword tag nlist
      ;
rword : TOKEN
      | LEFT
      | RIGHT
```

```
      | NONASSOC
      | TYPE
      ;
tag   : /* Empty: union tag ID optional. */
      | '<' IDENTIFIER '>'
      ;
nlist : nmno
      | nlist nmno
      ;
nmno  : IDENTIFIER       /* Note: literal invalid with % type. */
      | IDENTIFIER NUMBER  /* Note: invalid with % type. */
      ;


/* Rule section */


rules : C_IDENTIFIER rbody prec
      | rules  rule
      ;
rule  : C_IDENTIFIER rbody prec
      | '|' rbody prec
      ;
rbody : /* empty */
      | rbody IDENTIFIER
      | rbody act
      ;
act   : '{'
        {
          /* Copy action, translate $$, and so on. */
        }
        '}'
      ;
prec  : /* Empty */
      | PREC IDENTIFIER
      | PREC IDENTIFIER act
      | prec ';'
      ;
```

I mentioned in the introduction that you could have a parser generator generate its own parser. The grammar given above is what you would need in order to do that. Perhaps surpringly its grammar is no more complicated than that of the simple calculator `bc`.

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 insure 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 in the language used by `metamath`, one

of the more popular proof checkers:

..3 $\vdash S = n \in \mathbb{N} \mid (1 < n \wedge \forall m \in \mathbb{N}((n/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = n)))$

.3 $\vdash n \in \mathbb{N} \mid (1 < n \wedge \forall m \in \mathbb{N}((n/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = n))) \subseteq \mathbb{N}$

2 $\vdash S \subseteq \mathbb{N}$

...5 $\vdash (j \in \mathbb{N} \rightarrow \exists k \in \mathbb{N}(j < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k))))$

.....11 $\vdash (j \in \mathbb{N} \rightarrow 1 \leq j)$

.....10 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow 1 \leq j)$

.....11 $\vdash (j \in \mathbb{N} \rightarrow j \in \mathbb{R})$

.....11 $\vdash (k \in \mathbb{N} \rightarrow k \in \mathbb{R})$

.....12 $\vdash 1 \in \mathbb{R}$

.....12 $\vdash ((1 \in \mathbb{R} \wedge j \in \mathbb{R} \wedge k \in \mathbb{R}) \rightarrow ((1 \leq j \wedge j < k) \rightarrow 1 < k))$

.....11 $\vdash ((j \in \mathbb{R} \wedge k \in \mathbb{R}) \rightarrow ((1 \leq j \wedge j < k) \rightarrow 1 < k))$

.....10 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow ((1 \leq j \wedge j < k) \rightarrow 1 < k))$

.....9 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow (j < k \rightarrow 1 < k))$

.....8 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow (j < k \rightarrow (j < k \wedge 1 < k)))$

.....7 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow ((j < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k))) \rightarrow ((j < k \wedge 1 <$

.....7 $\vdash (((j < k \wedge 1 < k) \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k))) \leftrightarrow (j < k \wedge (1 < k \wedge \forall m \in$

....6 $\vdash ((j \in \mathbb{N} \wedge k \in \mathbb{N}) \rightarrow ((j < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k))) \rightarrow (j < k \wedge (1 <$

...5 $\vdash (j \in \mathbb{N} \rightarrow (\exists k \in \mathbb{N}(j < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k))) \rightarrow \exists k \in \mathbb{N}(j < k \wedge ($

..4 $\vdash (j \in \mathbb{N} \rightarrow \exists k \in \mathbb{N}(j < k \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \rightarrow (m = 1 \vee m = k)))))$

.....9 $\vdash (n = k \rightarrow (1 < n \leftrightarrow 1 < k))$

.....12 $\vdash (n = k \rightarrow (n/m) = (k/m))$

.....11 $\vdash (n = k \rightarrow ((n/m) \in \mathbb{N} \leftrightarrow (k/m) \in \mathbb{N}))$

.....12 $\vdash (n = k \rightarrow (m = n \leftrightarrow m = k))$

.....11 $\vdash (n = k \rightarrow ((m = 1 \vee m = n) \leftrightarrow (m = 1 \vee m = k)))$

.....10 $\vdash (n = k \to (((n/m) \in \mathbb{N} \to (m = 1 \vee m = n)) \leftrightarrow ((k/m) \in \mathbb{N} \to (m = 1 \vee m = k))))$

.....9 $\vdash (n = k \to (\forall m \in \mathbb{N}((n/m) \in \mathbb{N} \to (m = 1 \vee m = n)) \leftrightarrow \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1$

.....8 $\vdash (n = k \to ((1 < n \wedge \forall m \in \mathbb{N}((n/m) \in \mathbb{N} \to (m = 1 \vee m = n))) \leftrightarrow (1 < k \wedge \forall m \in \mathbb{N}((k/$

.....7 $\vdash (k \in S \leftrightarrow (k \in \mathbb{N} \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k)))))$

....6 $\vdash ((k \in S \wedge j < k) \leftrightarrow ((k \in \mathbb{N} \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k)))) \wedge j <$

....6 $\vdash (((k \in \mathbb{N} \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k)))) \wedge j < k) \leftrightarrow (k \in \mathbb{N} \wedge ((1$

.....7 $\vdash (((1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k))) \wedge j < k) \leftrightarrow (j < k \wedge (1 < k \wedge \forall m \in$

....6 $\vdash ((k \in \mathbb{N} \wedge ((1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k))) \wedge j < k)) \leftrightarrow (k \in \mathbb{N} \wedge (j <$

...5 $\vdash ((k \in S \wedge j < k) \leftrightarrow (k \in \mathbb{N} \wedge (j < k \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k)))$

..4 $\vdash (\exists k \in S j < k \leftrightarrow \exists k \in \mathbb{N}(j < k \wedge (1 < k \wedge \forall m \in \mathbb{N}((k/m) \in \mathbb{N} \to (m = 1 \vee m = k)))))$

.3 $\vdash (j \in \mathbb{N} \to \exists k \in S j < k)$

2 $\vdash \forall j \in \mathbb{N} \exists k \in S j < k$

2 $\vdash ((S \subseteq \mathbb{N} \wedge \forall j \in \mathbb{N} \exists k \in S j < k) \to S \approx \mathbb{N})$

$\vdash S \approx \mathbb{N}$

You may find it quite difficult even to identify what statement is being proved. In fact this is a proof of the fact that there are infinitely many primes.

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.

A language for zeroeth order logic

The propositional calculus, often called zeroeth order logic, governs the use of logical connectives 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. We'll use lower case letters, starting with p for variables and single symbols for logicial 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, having learned our lesson from the last chapter, 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 because some authors use it as a Boolean constant, signifying the value "true", although we won't do that.

The weird symbols for logical operators are unfamilar 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 we won't introduce those.

Our grammar is then

```
%start statement

%%

statement  : expression
           ;

expression : variable
           | ( expression binop expression )
           | [ expression binop expression ]
           | { expression binop expression }
           | ( ¬ expression )
           | [ ¬ 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. Single characters are all terminal symbols.

`binop` is short for binary operator and includes the operators $\wedge$, $\vee$, $\supset$, $\barwedge$, $\veebar$, $\equiv$, $\not\equiv$, $\subset$. If they look like junk in the listing above then that's because few monospaced fonts include those glyphs.

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 occurences of that variable in the the statement, not just all in some particular expression occuring in the statement.

There are certain symbols which are not part of our language but which we will use for talking about the language. 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 incan 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.

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 mathe-

maticians 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 $((\neg P) \vee Q)$.

Like "$\supset$" the more exotic symbols are all expressible in terms of "$\wedge$", "$\vee$", and "$\neg$". $(P \barwedge Q)$ has the same meaning as $(\neg(P \wedge Q))$. $(P \trianglerighteq Q)$ 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.

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.

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. To illustrate this I'll use the statement $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ which I proved earlier by the method of truth tables. I'll first give a version without tableaux and then explain how tableaux can be used to organise the argument.

Suppose $\{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}$ is 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.

Analytic tableaux

It can be difficult in arguments like the one above to keep track of what's known and what isn't at each point in the argument. In fact the argument above wasn'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. 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.

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. 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,

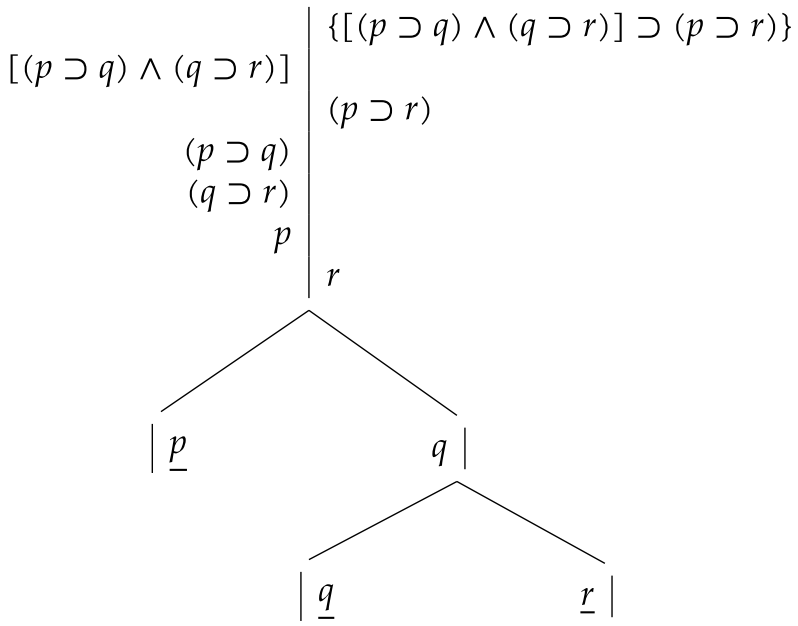
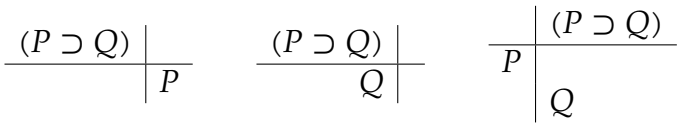
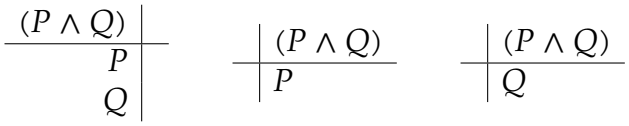


Figure 5: An analytic tableau for checking that $[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)$ is a tautology

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 expresssion was made, and which may appear on either side of the vertical bar. The diagrams for $\supset$ are

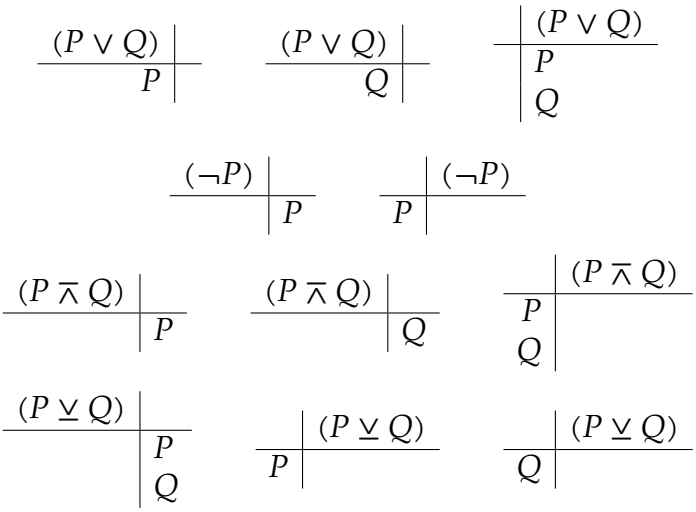


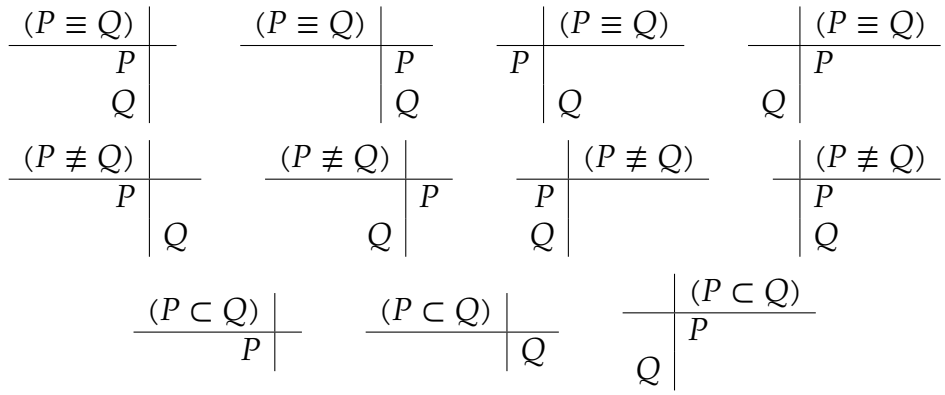
There are similar rules for $\wedge$.



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.

Another example

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.

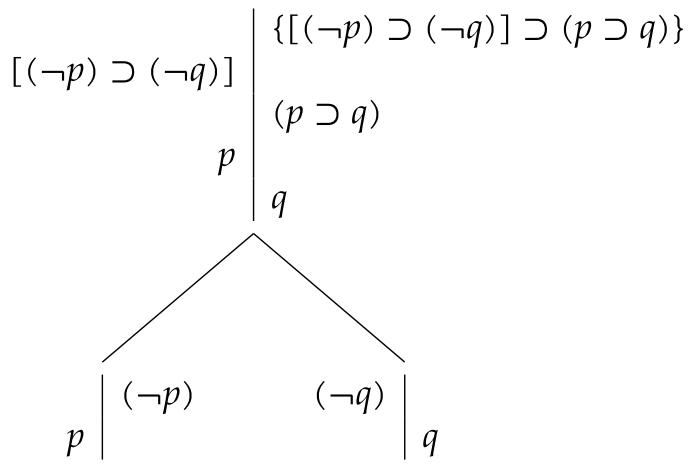


Figure 6: An analytic tableau to check whether $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is a tautology

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.

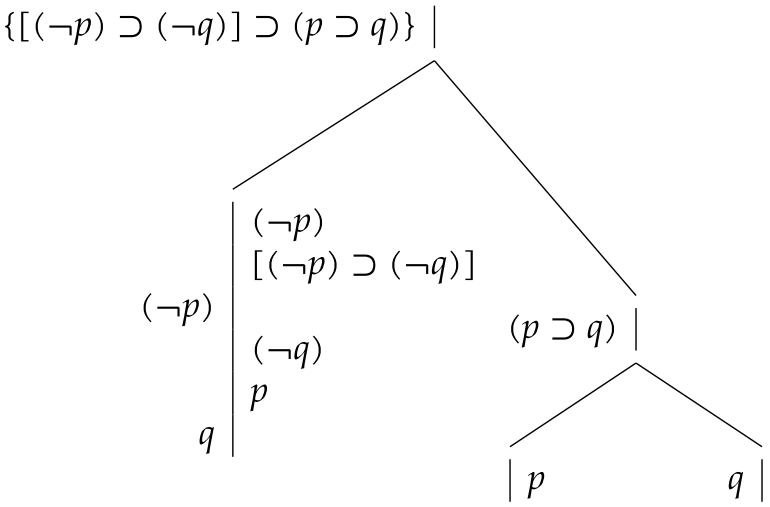


Figure 7: An analytic tableau to check whether $\{[(\neg p) \supset (\neg q)] \supset (p \supset q)\}$ is satisfiable

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 tree.

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.

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.

Tableaux as nondeterministic computations

The method of analytic tableaux is an example of nondeterministic computation, like the parsing method for context free languages we considered earlier. There's a specified initial state and at each point there is a set of operations available, specified by the tableaux rules, but no particular order of operations is specified. The situation for tableaux is better than for context free languages. Different orders of operations will give different tableaux, but no matter which order we choose we will reach an end state where either all branches have been closed or there is an open branch where all rules which can be applied have been applied. Which type of state we reach doesn't depend the order in which we apply the rules, although some orders may get us there faster than others. This is very different from the situation for context free grammars. There some orders may lead to a parse

tree while others will end with a list of tokens which don't match the input and still others won't terminate at all.

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 $\overline{\wedge}$ 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 are two rules of inference. The first is the rule of substitution discussed earlier. More explicitly, from any statement we can derive another statement by replacing each occurence of one of the variables with an expression. In fact we can do this for each variable in the statement. The second is that from two statements of the form P and $[P \mathbin{\overline{\wedge}} (Q \mathbin{\overline{\wedge}} R)]$ we can derive the statement R.

Soundness of the Nicod system

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 two rules of inference of the system have the property that if applied to tautologies they lead to a tautology. In fact the first is, as discussed earlier, a general property of zeroeth order logic and the second 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	F

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 sense.

Completeness of the Nicod system

We just saw that the Nicod system is sound, which in this case means that every theorem is a tautology. It's also complete, in the sense that every tautology is a theorem. As you might imagine, proving this is rather painful.

The fact that we have only one axiom and two rules of inference made proving soundness relatively easy but it makes proving completeness very hard. We won't even attempt it.

It's important to note that soundness and completeness are properties of the system and its interpretation. They express, respectively, the fact that every provable statement is true and every true statement is provable. Provability is a concept within the system, but truth depends on the interpretation given to statements, which is not part of the system.

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. 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 whose language 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. 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.
4. From P and $(P \supset Q)$ we can deduce Q.
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.
9. The "Rule of Fantasy", to be described below.
10. The "Rule of Substitution", subject to restrictions to be discussed below.

The first four 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. Similarly the third 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. The fourth rule is one we've met before, under the name modus ponens. 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 practise 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 contradition, 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 tableax 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 was introduced and before it was 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 contradic-

tion. 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

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

$$
\begin{array}{ll}
. & p \\
. & . \quad q \\
. & . \quad (p \wedge q) \\
. & [q \supset (p \wedge q)] \\
\{p \supset [q \supset (p \wedge q)]\}
\end{array}
$$

Our first step is 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

$$
\begin{array}{l}
. \quad (\neg p) \\
[(\neg p) \supset (\neg p)] \\
[p \vee (\neg p)]
\end{array}
$$

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. 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 both the Nicod and Łukasiewicz systems. 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:

$$\begin{array}{ll} . & p \\ . & q \\ & (p \supset q) \end{array}$$

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 substitutible 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.

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.

More proofs

Here's another fairly short proof. Try going through each line and seeing if you can identify which rule is being used.

$$
\begin{array}{l}
.\quad (p \supset q) \\
.\quad [(\neg q) \supset (\neg p)] \\
.\quad [q \vee (\neg p)] \\
\{(p \supset q) \supset [q \vee (\neg p)]\}
\end{array}
$$

The next proof is a bit trickier to find but is very useful.

$$
\begin{array}{l}
.\quad [p \wedge (\neg p)] \\
.\quad p \\
.\quad (\neg p) \\
.\quad .\quad (\neg q) \\
.\quad .\quad [\neg(\neg p)] \\
.\quad \{(\neg q) \supset [\neg(\neg p)]\} \\
.\quad [(\neg p) \supset q] \\
.\quad q \\
\{[p \wedge (\neg p)] \supset q\}
\end{array}
$$

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

Here are two more complicated examples.

$$
\begin{array}{l}
\quad . \quad [\neg(p \supset q)] \\
\quad . \quad [\neg(\{\neg q\} \supset \{\neg p\})] \\
\quad . \quad [\neg(q \vee \{\neg p\})] \\
\quad . \quad [(\neg q) \wedge (\neg\{\neg p\})] \\
\quad . \quad [(\neg q) \wedge p] \\
\{[\neg(p \supset q)] \supset [(\neg q) \wedge p]\}
\end{array}
$$

and

$$
\begin{array}{l}
. \quad [p \wedge (\neg p)] \\
. \quad p \\
. \quad (\neg p) \\
. \quad . \quad (\neg q) \\
. \quad . \quad [\neg(\neg p)] \\
. \quad \{(\neg q) \supset [\neg(\neg p)]\} \\
. \quad [(\neg p) \supset q] \\
. \quad q \\
\{[p \wedge (\neg p)] \supset q\} \\
\{[r \wedge (\neg r)] \supset r\} \\
. \quad \{(p \vee q) \wedge [(p \supset r) \wedge (q \supset r)]\} \\
. \quad (p \vee q) \\
. \quad [(p \supset r) \wedge (q \supset r)] \\
. \quad (p \supset r) \\
. \quad (q \supset r) \\
. \quad [(\neg p) \supset q] \\
. \quad . \quad (\neg p) \\
. \quad . \quad q \\
. \quad . \quad r \\
. \quad [(\neg p) \supset r] \\
. \quad [(\neg r) \supset p] \\
. \quad . \quad (\neg r) \\
. \quad . \quad p \\
. \quad . \quad r \\
. \quad [(\neg r) \supset r] \\
. \quad r \\
(\{(p \vee q) \wedge [(p \supset r) \wedge (q \supset r)]\} \supset r)
\end{array}
$$

You might notice that the first part simply repeats the proof given earlier for $\{[p \wedge (\neg p)] \supset q\}$. One annoying feature of formal systems is that they

start the proof of every theorem from scratch. No mechanism is provided for reusing previously proved theorems other than repeating their proofs.

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

$$
\begin{array}{ll}
. & [(p \supset q) \wedge (q \supset r)] \\
. & . \quad p \\
. & . \quad (p \supset q) \\
. & . \quad (q \supset r) \\
. & . \quad q \\
. & . \quad r \\
. & (p \supset r) \\
 & \{[(p \supset q) \wedge (q \supset r)] \supset (p \supset r)\}
\end{array}
$$

Soundness, consistency and completeness

Because natural deduction systems were developed to mimic exactly the sort of reasoning we formalised with the method of analytic tableau it is possible, though somewhat tedious, to give an algorithm for converting a tableau into a proof in this system. The construction of a tableau for a tautology can also be automated. Combining these we see that there is an algorithm which takes tautologies and generates formal proofs for them. So every tautology is a theorem and therefore the system is complete. It's also sound because all the axioms are true and all the rules of inference preserve truth. It's easy to see that all the axioms are true because there aren't any! Showing that the rules of inference are sound is fairly straightforward until we get to the rule of fantasy and the rule of substitution. You can read through the descriptions of those rules and convince yourself that they can't be used to prove true statements from untrue ones. There's a good chance you could have convinced yourself even if I'd left out the restrictions on substitution though and neglected to mention that the unrestricted version makes $p \supset q$ into a theorem, so I'm not sure how far you can trust your intuition on these things. Unfortunately I can't give a formal proof of soundness though because we don't have a formal language in which we can state it. Soundness, as before, implies consistency.

It's also possible for the standard axiomatic systems, like that of Nicod or Łukasiewicz, to give algoritms for converting proofs in a natural deduction system to proofs within the axiomatic system. This is generally the

easiest way of showing that those systems, which we already know to be consistent, are complete.

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 incompatible 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. It's clear that any statement about the integers can be made up of such predicates, together with quantifiers.

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 2023 is an integer, so we are assured by this statement that there is a prime number greater than 2023. 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.

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, beyond a hidden assumption that, whatever they are, there is at least one of them. It is possible to dispense with even this assumption, and would probably be a good idea to do so, with what's called inclusive logic, but this has never been particularly popular.

A language for first order logic

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

```
%start statement

%%

statement      : expression
               ;

expression     : atomic_expression
               | ( expression binop expression )
               | [ expression binop expression ]
               | { expression binop expression }
               | ( ¬ expression )
               | [ ¬ expression ]
               | { ¬ expression }
               | ( quantifier variable . expression )
               | [ quantifier variable . expression ]
               | { quantifier variable . expression }
```

```
              ;

atomic_expression : ( atom )
                  | [ atom ]
                  | { atom }
                  ;

atom          : predicate
              | atom variable
              | atom paramater
              ;

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 expresions 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 occurences 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 promise 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.

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 non-empty set, called the "domain". Then we assign an element of that set to each variable or parameter. To each predicate we assign a relation, which you can safely think of as a Boolean-valued function. 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.

Once we've done this we can assign truth values to expressions by starting with atomic expressions and combining them to form larger and larger expressions, much as we did in zeroeth order logic. There are some differences though. The values we assigned to the variables when constructing our interpretation are only to be used for free occurrences of those variables.

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, 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) 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 both false. 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 for first order logic

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 quantifiers.

$$\frac{[\forall V.(PV)]}{(PA)} \qquad \frac{[\exists V.(PV)]}{(PB)}$$

$$\frac{}{\begin{array}{|l}[\forall V.(PV)]\\(PB)\end{array}} \qquad \frac{}{\begin{array}{|l}[\exists V.(PV)]\\(PA)\end{array}}$$

The notational conventions are as discussed previously but in this case A stands for any parameter but B stands for any new parameter, i.e. one which has not been used previously in the tableau. PV stands for any expression

and PA or PB stand for the same expression but with all free occurrences of the variable denoted by V replaced by the parameter denoted by A or the parameter denoted by B, respectively.

So if we have the expression $[\forall x.(Fx)]$ on the left hand side of the vertical bar we'd be allowed to write (Fa) to the left of the bar. We'd also be allowed to write (Fb) or F followed by any other parameter. But this rule isn't restricted to unary predicates. It applies to any expression. If, for example, we had the expression to the left of the bar $\{\forall x.[(gwx) \supset (hxy)]\}$ then we could write $[(gwa) \supset (hay)]$ to the left. This is the tableau counterpart of saying "We know that $[(gwx) \supset (hxy)]$ for all x so in particular $[(gwa) \supset (hay)]$."

The rule for a $\exists$ on the left is similar, but the counterpart in an informal proof is now "We know there is at least one x such that $[(gwx) \supset (hxy)]$. Let a be such an x. Then $[(gwa) \supset (hay)]$." This explains reason for the restriction to new parameters, which we already met in the second of our two informal proofs above. As another example of this phenomenon, if we have the statements "there is at least one even number" and "there is at least one one odd number" then we can legitimately say "Let a be an even number and let b be an odd number". We couldn't legitimately say "Let a be an even number and let a be an odd number" and then conclude that some number is simultaneously even and odd.

The rules to the right of the bar are similar. The only way a universal statement can be false is if there is at least one counterexample. The corresponding tableau rule allows us to name that counterexample with a parameter. It has to be a new parameter because we have no right to assume it is equal to any previously named value. For a universal statement there is no such restriction. If a universal statement is false then every instance of it is false, so it will be false for any previously named value as well.

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.

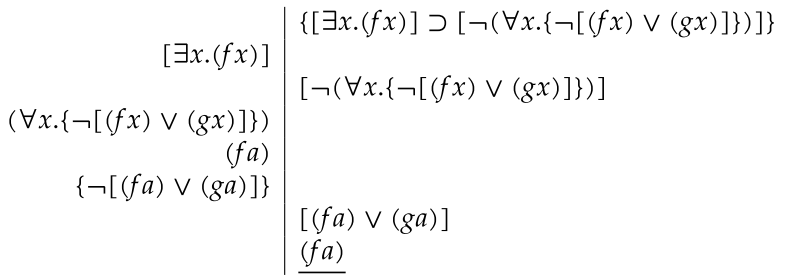


Figure 8: Tableau to check that {[∃x.(fx)]⊃[¬(∀x.{¬[(fx)∨(gx)]})]} is valid

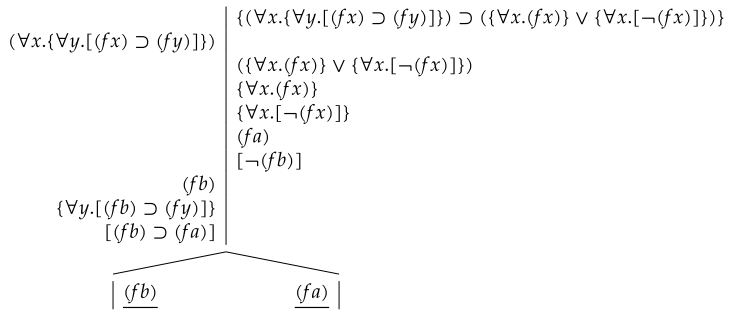


Figure 9: Tableau to check that {(∀x.{∀y.[(fx)⊃(fy)]})⊃({∀x.(fx)}∨{∀x.[¬(fx)]})} is valid

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

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.

We need four rules of inference, which correspond to the four tableau rules. The ones corresponding to the two unrestricted tableau rules are mostly unproblematic. One rule says that from a statement of the form $[\forall V.(PV)]$ we can deduce one of the form (PA) and the other says that from a statement of the form (PA) we can deduce one of the from $[\exists V.(PV)]$. The first of these is an exact translation of one of our tableau rules and the second can be thought of as the contrapositive of another of our rules.

There is one subtlety of these rules which should be pointed out. Applying those two rules in the order stated takes us from $[\forall V.(PV)]$ to (PA) to $[\exists V.(PV)]$. If we allowed interpretations where the domain is empty then these rules would be unsound. $[\forall V.(PV)]$ would always be true, no matter which variable and expression V and P represent because a universal statement is true unless there is some counterexample in the domain and an empty domain can contain no counterexample. On the other hand $[\exists V.(PV)]$ is false, again no matter which variable and expression V and P represent, because an existential statement can only be true if there is an example in the domain and there can be no example in an empty domain.

If you want something like first order logic but which is applicable to domains which are not known in advance to be non-empty then you need inclusive logic, which has a different set of rules of inference for quantifiers, rather than first order logic. Inclusive logic is also known as universally free logic or just free logic, although the latter name is dangerous because is is also used for something intermediate between inclusive logic and first order logic.

Besides not assuming the domain is non-empty inclusive logic also allows for parameters which might not refer to any element of the domain. This is more useful than it sounds. You can't safely apply first order logic to a language which includes phrases like "the current king of France" but it's hard

to build a language which excludes such phrases but includes "the current king of Norway", who does exist. English certainly makes no attempt to do prevent references to things which don't exist and therefore essentially all "real world" examples you find in logic textbooks are suspect. Standard mathematical language also allows provides names for non-existent objects, like the quotient of a number by zero, the value of a function at a point not in its domain, the limit of a sequence which doesn't converge, etc. Mathematicians go ahead and apply first order logic to mathematics anyway, and in fact I'll do so in the next chapter with elementary arithmetic. That particular application will be harmless because the particular language I'll use for arithmetic won't allow references to non-existent objects. It will have, for example, a symbol for multiplication but none for division. It's hard to do that in any setting more complicated than elementary arithmetic though so there is a real mismatch between modern mathematical language and the first order logic which supposedly forms the basis of mathematical reasoning.

The two rules of inference above are still relatively unproblematic compared to the other two quantifier rules though. The rules in question are variants of the rule of fantasy, but one of them doesn't introduce any hypothesis! Roughly we would like to say, for example, that if we enter a new scope and manage to derive the statement (PA) within it then we can leave that scope and have the statement $[\forall V.(PV)]$ in the outer scope. The idea underlying this is that if we can prove Pa without making any assumptions on a then P should be true universally. The problem with this is similar to the one we encountered in zeroeth order logic with the rule of substitution and the solution is somewhat similar. We have to restrict our use inside the scope of variables from outside the scope. In this case we need to avoid using statements in which the parameter a appears. Otherwise we've not kept to the "without making any assumptions on a" part of the justification for the rule. Or we could allow the use of such statements, but with the requirement that all occurrences of a be replaced by some new parameter.

There is a similar rule of inference corresponding to our remaining tableau rule. If we have a statement in scope of the form $[\exists V.(PV)]$ then this rule allows us to introduce a hypothesis (PA). When we discharge this hypothesis we can conclude $[(PA) \supset Q]$, provided two restrictions are met. The first is the same as in the previous rule: the variable represented by V should not occur in any of the statements we use from any wider scope. The second

restriction is that it should also not occur in Q.

Soundness, consistency and completeness of first order logic

The formal system described above is sound for any interpretation of the type discussed earlier. This implies consistency as well. It is also complete. The easiest way to prove this is by giving an algorithm for converting a closed tableau into a proof in the natural deduction system. We've already seen how to find, in finite time, a closed tableau for any valid statement so every valid statement has a proof. If you like complete systems you should take a moment to enjoy this fact before reading further. This is the last complete system we will see.

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 : ( ¬ bool_exp) | ( bool_exp b_operator bool_exp )
         | ( quantifier variable . bool_exp )
         | ( num_exp relation num_exp ) ;
         | [ ¬ bool_exp) | [ bool_exp b_operator bool_exp )
         | [ quantifier variable . bool_exp )
         | [ num_exp relation num_exp ) ;
         | { ¬ bool_exp} | { bool_exp b_operator bool_exp }
```

```
        | { quantifier variable . bool_exp }
        | { num_exp relation num_exp } ;
b_operator : ∧ | ∨  | ⊃ ;
quantifier : ∀ | ∃ ;
variable : letter | variable ! ;
variable : v | w | x | y | z ;
relation : = | ≤ ;
num_exp : 0 | variable | num_exp ' | ( num_exp a_operator num_exp ) ;
a_operator : + | · ;
```

There are now two types of expressions, boolean and numerical. The numerical ones are meant to have natural numbers as values. The ′ takes a numerical expression and increments it. We sometimes refer to x' as the successor of x.

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!

It would have been possible to use decimal or binary representations directly 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.

New to this module, but presumably familiar, are the binary relations $=$ and $\leq$. They take a pair of numerical expressions and give a Boolean expression. This is unlike the Boolean operators, which combine Boolean expressions to give a Boolean expression or the arithmetic operators $+$ and $\cdot$ which combine numerical expressions into numerical ones.

Other than this our language for arithmetic is borrowed from first order logic, but the role of predicates is now played by Boolean expressions and the role of parameters is played by numerical expressions.

If you're wondering why the subtraction and division operators are missing, that's a consequence of using first order logic. First order logic, as mentioned earlier, does not cope well with names for non-existent objects. We would quickly encounter problems if we allowed expressions like $0 - 0'$ or $0'/0$ into our language. Of course $0 - 0'$ exists, but not as a natural number.

There is some redundancy in this language. 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 $x + y = z$.

Expressing more complex ideas

All of elementary arithmetic can be expressed in this language, but sometimes a bit of ingenuity is required. Sometimes this is relatively straightforward. We don't have a $<$ sign, for example, but we can still express the idea that $x < z$. In fact we can do so in many different ways, including

- $(x' \leq z)$,
- $\{(x \leq z) \wedge [\neg(x = z)]\}$,

- $[\neg(z \leq x)]$, and
- $\{\exists y.[(x + y') = z]\}$.

We can also 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:

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

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 or equal to 2.

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 or equal to w. In our language this is

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

Arithmetic subsets

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 x 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.

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. 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?

It would be nice to have a general principle saying that, if we can define a sequence within our language, for example by specifying the initial element and a rule for getting from each element to the next, like "start from 1 and keep multiplying by 6", then the set of values should be arithmetic. It's possible to do this, but it requires a great deal of preliminary work.

Encoding

We can encode any formal language into natural numbers. There are in fact a number of ways to do this. The simplest, which works when the number of tokens is finite, is to let b be the number of tokens, assign each token to one of the "digits" $1, 2, \ldots, b$, and use the base $b + 1$ representation of natural numbers. In more detail, given an element of the given language, i.e. a

list of tokens which belongs to the language, we replace each token by the corresponding digit and view the result as a natural number represented in base $b + 1$. In the other direction, given a natural number we can form its base $b + 1$ representation, which is a list of digits, and, if the digit 0 does not appear, we can replace those digits by the corresponding tokens, obtaining a list of tokens. This list may or may not be an element of the language. This is, of course, the reverse of the process by which we generated natural numbers from lists of tokens. In this way we can identify the language with a subset of the natural numbers.

You may wonder why I used base $b + 1$ and avoided using the 0 digit in the encoding above. The reason is technical. If we allowed 0 and the corresponding token is one which could occur as the initial token in an element of the language then the natural number representation of that string would start with a 0. Removing that 0 would give me the same natural number, but corresponding to a shorter list of tokens, which might or might not be an element of the language but certainly isn't the same element of the language. Our encoding therefore would be lossy; it would be impossible to recover lists of tokens uniquely from natural numbers. Since we can't use 0 as a digit we have to use base $b + 1$ rather than base b.

As an example of the encoding above, consider the element `(()(()))` in the language of balanced parentheses. This language has two tokens so we we use base 3. Associating `(` to 1 and `)` to 2 we replace `(()(()))` by 11211222, which is the base 3 representation of the number whose decimal representation is 3536. So this number is the encoding of `(()(()))`.

The method above can be modified to cope with languages with infinitely many tokens, provided the set of tokens is countable, a term which will be defined in the next chapter. All the languages we'll encode will have only finitely many tokens though.

There are other ways to encode languages as natural numbers. The precise encoding chosen doesn't affect the validity of anything I'll write below, although some encodings make proofs easier and some make them harder. The particular encoding described above is simple but tends to make proofs hard. I generally won't be providing proofs though so it's not worth the effort of setting up an encoding which is better adapted providing proofs.

The most important thing to know about encodings of languages is that

if the language is defined by a phrase structure grammar then the set of natural numbers corresponding to lists in the language is arithmetic. This is quite painful to prove, unfortunately.

Encoding arithmetic in arithmetic

The construction described above is applicable to any context free language. Our language for arithmetic is a context free language so we can encode it in the manner just described. This means we are encoding statements about the natural numbers as natural numbers.

There are a number of useful things this allows us to do. One is to implement the idea discussed at the end of the section on arithmetic sets of showing that any sequence we can define in our language gives rise to an arithmetic set. The key idea is that encoding gives us a way to represent the concept "can define in our language" within the our language. Previously we've scrupulously maintained the distinction between statements within a language and statements about that language, and that distinction remains important, but in the case of arithmetic we can now start finding proxies within the language for statements about the language.

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 the first person to introduce such a system. The particular version used below is essentially that of Hofstadter.

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.

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.

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. $0''$.

Other than the two 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 five axioms for arithmetic:

1. $\{\forall x.[\neg(x' = 0)]\}$
2. $\{\forall x.[(x + 0) = x]\}$

3. $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
4. $\{\forall x.[(x \cdot 0) = 0]\}$
5. $[\forall x.(\forall y.\{(x \cdot y') = [(x \cdot y) + x]\})]$

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 says that there's no natural number which, when incremented, gives 0, i.e. that 0 is not the successor of any natural number. The second 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 third 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 second and third 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 fourth 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 fourth and fifth axioms are essentially a recursive definition of multiplication. The fourth axiom tells us how to multiply by 0 and the fifth axiom allows us to get, one step at a time, from multiplication by 0 to multiplication by any natural number.

There are also some arithmetic rules of inference.

1. From a statement of the form $(X = Y)$ we can deduce $(Y = X)$.
2. From statements of the form $(X = Y)$ and $(Y = Z)$ we can deduce $(X = Z)$.
3. From a statement of the form $(X = Y)$ we can deduce $(X' = Y')$.
4. From a statement of the form $(X' = Y')$ we can deduce $(X = Y)$.
5. 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 two rules of inference say that equality is reflexive and transitive. The third is a special case of a general principle that if two quantities are equal then the results of applying the same operation to both are also equal.

The particular special case is that where the operation in question is incrementing. The converse of the general principle is not true in general. It's not true, for example, that if two numbers give the same product when multiplied by 0 then they are equal. The converse does hold for incrementing though: if two numbers have the same successor then they are equal.

Induction

The fifth 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 re-

placed 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.

A formal proof

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. $\{\forall x.[(x + 0) = x]\}$
2. $[(0'' + 0) = 0'']$
3. $[(0'' + 0)' = 0''']$
4. $(\forall x.\{\forall y.[(x + y') = (x + y)']\})$
5. $\{\forall y.[(0'' + y') = (0'' + y)']$
6. $[(0'' + 0') = (0'' + 0)']$
7. $[(0'' + 0') = 0''']$
8. $[(0'' + 0')' = 0'''']$
9. $[(0'' + 0'') = (0'' + 0')']$
10. $[(0'' + 0'') = 0'''']$

The first and fourth lines are our second and third arithmetic axioms. The second line is the result of one of our logical rules of inference, eliminating the universal quantifier from the first line and replacing the associated variable with the numerical expression $0''$. Similarly the fifth line is derived from the fourth by eliminating the universal quantifier and replacing the variable by $0''$. Both the sixth line and the ninth are derived from the fifth by removing the universal quantifier and replacing the variable, in one case by 0 and in the other by $0'$. The third line is derived from the second by applying our fourth arithmetic rule of inference and the eighth line is similarly obtained from the seventh. The seventh line is derived from the third and sixth by means of our second arithmetic rule of inference, and the tenth is similarly derived from the eighth and ninth.

Statements more complicated than $2 + 2 = 4$ have correspondingly longer proofs. Hofstadter, for example, gives a proof that addition is commutative. It's 56 lines long. 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 and Tarski's theorem.

I've already mentioned that formal languages described by phrase structure grammars have encodings which are arithmetic sets. If we have axioms and rules of inference as well then we can look at the sublanguage consisting of theorems in this formal system. The set of the encodings of theorems also turns out to be arithmetic. This is a theorem of Gödel. The word "theorem" is used in different senses in the two preceding sentences. In the first of them it means a statement with a formal proof and in the second it means a statement with an informal proof. Gödel's theorem applies to Peano arithmetic, but is more general than that.

With an interpretation which enables us to characterise statements as true of false we can also look at the sublanguage of true statements. In particular we can consider the encodings of true statements in Peano arithmetic. This

set is not arithmetic. That is a theorem of Tarski.

Gödel's theorem is based on the insight that it's not only possible, as we've already discussed, to make statements about Peano arithmetic within Peano arithmetic but it's even possible to construct statements in Peano arithmetic which refer to themselves. In essence you can create statements with interpretations like "I cannot be proved" or "if I can be proved then I can also be disproved". The first of these sentences is roughly the one at the heart of Gödel's argument. The second is used in a strengthened version of that theorem due to Rosser.

Combining the theorems of Gödel and Tarski we see that the set of theorems in Peano arithmetic and the set of true statements cannot be the same set. The situation is even worse than it appears though. This is not simply a problem with a particular formal system for arithmetic, which we might hope to fix by, for example, adding a missing axiom or rule of inference. Gödel's theorem is not specific to a particular formal system but applies to any formal system. Tarski's theorem doesn't even mention formal systems. It is purely a statement about the language of true statements. So no matter what replacement formal system we consider for arithmetic the set of true statements and theorems will always be different.

The combination of Gödel and Tarski tells us there must be a true statement which is not a theorem or a theorem which is not a true statement. There could also be both. The theorems don't provide any insight into which of these possibilities occurs. A false theorem would be much more damaging than a true but unprovable statement. Most mathematicians and logicians believe that all theorems in Peano arithmetic are true but that there are true statements which are not theorems. In other words, they believe the system is sound but incomplete. Other than the fact that no one has yet found a contradiction there is no actual evidence for this belief. Gödel himself was skeptical of the soundness of arithmetic. His theorem was the result of an ultimately unsuccessful attempt to prove the inconsistency of Peano arithmetic.

Although he didn't succeed in proving the inconsistency of arithmetic Gödel did succeed in killing the project, dating back to Euclid, of fully axiomatising all of mathematics. Gödel's work is generally remembered in connection with Peano arithmetic but in fact he considered two axiomatic theories which seemed like candidates for the axiomatisation of mathe-

matics. One was Peano arithmetic and the other was axiomatic set theory, which is our next topic.

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 : /* empty */ | 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 | K | L | 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. It's possible to do first order logic with bounded quantifiers, provided all quantifiers are over the same set and this 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 elements. $\bigcap$ and $\bigcup$ indicate intersection and union, respectively. These are each used in two different ways. When not immediately preceded by a set expression $\bigcup$ is an operator which takes a set and gives you its union, i.e. the set whose elements are the elements of its elements. In other words, $[x \in [\bigcup A]]$ if and only if there is some B such that x is an element of B and B is an element A. The most common case of unions is one where the set of sets has two elements so we have a special notation for this. We write $[B \bigcup C]$ to mean $[\bigcup\{B, C\}]$, i.e. the set of all elements which are in B or 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 and element 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. $[B \bigcap C]$ means $[\bigcap\{B, C\}]$. Our other two set operations are $\setminus$ for the relative complement and $\times$ for the Cartesian product. $[A \setminus B]$ is the set of elements of A which are not in B, i.e. $\{x \in A \colon [\neg[x \in B]]\}$. $[A \times B]$ is the set of pairs (x, y) where x is an element of A and y is an element of B. $[PA]$ is the power set of A, i.e. the set of all subsets 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 mosttly 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. What follows is essentially the set theory of Zermelo, with one axiom removed.

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 element of A is an element of B and vice versa then $A = B$.
- Elementary sets: $\emptyset$ is a set. For all x we have $[\neg[x \in \emptyset]]$. For all x we have a set $\{x\}$, of which x is an element and there are no other elements. Similarly, for all x and y we have a set $\{x, y\}$ such that x and y are elements and there are no other elements.
- Separation: For every variable x, set A and Boolean expression θ the set $\{x \in A : \theta\}$, whose elements are those elements 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 element of B is an element 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 their 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 gives $[[\bigcap A] = \emptyset]$. This has some unfortunate consequences, but fewer such consequences than any other definition. Really, the only reason we define the intersection at all in this case is that first order logic can't cope with expressions which don't denote anything in the domain. In general if you're taking an intersection of a set of sets then you should probably add a hypothesis that that set of sets is non-empty.

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.

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. Suppose that for all members A of a set B x is a member of A. Does it follow that $[x \in [\bigcap B]]$? Yes, if B is a non-empty set of sets. No, if B is the empty set. In that case every x would vacuously satisfy the condition that for all members A of a set B x is a member of A, but we can't have a set which contains all x. The options are to leave $[\bigcap \emptyset]$ undefined or to impose a restriction as above. Since first order logic can't cope with expressions whose value is undefined we have to impose the restriction. This restriction then propagates to a number of other statements. For example, it's true that if $[A \subseteq B]$ then $[[\bigcap B] \subseteq [\bigcap A]]$, but only under the restriction that A is a non-empty set of sets.

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.
- $[[x \in [A \bigcap B]]$ if and only if $[[x \in A] \wedge [x \in B]]]$. This is more or less the definition of the $\bigcap$ operator.
- $[[x \in [A \bigcup B]]$ if and only if $[[x \in A] \vee [x \in B]]]$. This is more or less the definition of the $\bigcup$ 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 $\bigcap$, $\bigcup$ and $\setminus$ are expressible in terms of the four Boolean operators $\wedge$, $\vee$, $\neg$, and $\supset$. $\bigcap$ corresponds to $\wedge$ and $\bigcup$ 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 $[[[A \bigcup B] \bigcup C] = [A \bigcup [B \bigcup C]]]$. $[[p \vee [q \vee r]] \supset [[p \vee q] \vee r]]$ 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 $[x \in [B \bigcup C]]$ and $[[x \in A] \vee [x \in B]]$ with $[x \in [A \bigcup B]]$, so

$$[[[x \in A] \vee [x \in [B \bigcup C]]] \supset [[[x \in [A \bigcup B]]] \vee [x \in C]]].$$

Then we can replace $[[x \in A] \vee [x \in [B \bigcup C]]]$ by $[x \in [A \bigcup [B \bigcup C]]]$ and $[[[x \in [A \bigcup B]]] \vee [x \in C]]$ by $[x \in [[A \bigcup B] \bigcup C]]$, so

$$[[x \in [A \bigcup [B \bigcup C]]] \supset [x \in [[A \bigcup B] \bigcup C]]]$$

and hence

$$[[A \bigcup [B \bigcup C]] \subseteq [[A \bigcup B] \bigcup C]].$$

Similarly, $[[[p \vee q] \vee r] \supset [p \vee [q \vee r]]]$ is a tautology so

$$[[[A \bigcup B] \bigcup C] \subseteq [A \bigcup [B \bigcup C]]].$$

Combining that with the inclusion already obtained gives

$$[[[A \bigcup B] \bigcup C] = [A \bigcup [B \bigcup C]]].$$

The following facts about sets can similarly be proved using tautologies borrowed from zeroeth order logic.

- $[[A \bigcap B] \subseteq A]$
- $[[A \bigcap B] \subseteq B]$
- $[A \subseteq [A \bigcup B]]$
- $[B \subseteq [A \bigcup B]]$
- $[[A \setminus B] \subseteq A]$
- $[[A \bigcap A] = A]$
- $[[A \bigcup A] = A]$
- $[[A \bigcap B] = [B \bigcap A]]$

- $[[A \bigcup B] = [B \bigcup A]]$
- $[[[A \bigcap B] \bigcap C] = [A \bigcap [B \bigcap C]]]$
- $[[[A \bigcup B] \bigcup C] = [A \bigcup [B \bigcup C]]]$
- $[[[A \bigcap [B \bigcup C]] = [[A \bigcup C] \bigcap [B \bigcup C]]]$
- $[[[A \bigcup [B \bigcap C]] = [[A \bigcap C] \bigcup [B \bigcap C]]]$
- $[[A \bigcap [A \bigcup B]] = A]$
- $[[A \bigcup [A \bigcap B]] = A]$
- $[[A \setminus [A \setminus B]] = [A \bigcap B]]$
- $[[C \setminus [A \bigcap B]] = [[C \setminus B] \bigcup [C \setminus A]]]$
- $[[C \setminus [A \bigcup B]] = [[C \setminus B] \bigcap [C \setminus A]]]$
- $[[A \setminus [B \setminus C]] = [[A \bigcap C] \bigcup [B \setminus C]$
- $[[[A \setminus B] \setminus C] = [A \setminus [B \bigcup C]]]$
- $[[[A \setminus B] \bigcap C] = [A \bigcap [C \setminus B]]]$

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, $P\emptyset = \{\emptyset\}$ and $PP\emptyset = \{\emptyset, \{\emptyset\}\}$. The only non-empty member of $PP\emptyset$ is $\{\emptyset\}$. It 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 $A \bigcap B$ is finite because $A \bigcap B \subseteq A$ and $A \bigcap B \subseteq B$. 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.

Now we turn our attention to unions. Suppose A and B are finite sets and that C is a non-empty set of subsets of $A \bigcup B$. Define

$$D = \{E \in PA : \exists F \in C : A \bigcap F = E\}.$$

In other words, D is the set of sets which are intersections of A with members of C. This is a set of subsets of A. It's a non-empty set because C is non-empty and taking a member of C and intersecting it with A gives a member of D. A is finite so D has a minimal member. Let G be such a

member. Now let

$$H = \{I \in PB : G \bigcup I \in C\}.$$

This is a set of subsets of B. Since $G \in D$ there must be some $F \in C$ such that $A \bigcap F = G$. Let $J = F \setminus A$ for such an F. Now $J \subseteq F$ and $F \in C$ and $C \in PP[A \bigcup B]$ so $J \subseteq A \bigcup B$. But no member of J is a member of A so they are all members of B. In other words $J \subseteq B$. Also $G \bigcup J = [A \bigcap F] \bigcup [F \setminus A]$ so $G \bigcup J = F$ and hence $G \bigcup J \in C$. Therefore $J \in H$ and so H is a non-empty set of subsets of B. B is finite so H must have a minimal member. Let K be such a member. Now $K \setminus A \in H$ and K was minimal so $K \setminus A = K$ and therefore $A \bigcap K = \emptyset$. I claim that $G \bigcup K$ is a minimal member of C. First of all, it is a member of C because $K \in H$ and the definition of H requires the union of any member of H with G to be in C. Suppose L is a member of C such that $L \subseteq G \bigcup K$. Then $A \bigcap L \subseteq A \bigcap [G \bigcup K]$. Also, $A \bigcap [G \bigcup K] = [A \bigcap G] \bigcup [A \bigcap K]$. Now $G \subseteq A$ so $A \bigcap G = G$ and $A \bigcap K = \emptyset$ so $A \bigcap L \subseteq G$. $L \in C$ so $A \bigcap L \in D$ and G was a minimal member of D so $A \bigcap L = G$. Then $G \bigcup [L \setminus A] = L$. Since $L \in C$ it follows that $L \setminus A \in H$. Now $L \setminus A \subseteq K$ and K is a minimal member of H so $L = A \subseteq K$. From $L = [A \bigcap L] \bigcup [L \setminus A]$ we see that $L = G \bigcup K$. In other words we've shown that if $L \in C$ and $L \subseteq G \bigcup K$ then $L = G \bigcup K$, i.e. that $G \bigcup K$ is a minimal member of C. C was an arbitrary non-empty set of subsets of $A \bigcup B$ so every such set of subsets has a minimal member. Therefore $A \bigcup B$ is finite.

From this and the fact that $\{x\}$ is finite we find that if A is finite then so is $A \bigcup \{x\}$ 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 $A \bigcup \{x\}$, 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 $C \bigcup \{x\} \in B$. Then $A \in B$.

To prove this, first set $D = B \bigcap PA$. 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 $F = E \bigcup \{x\}$. 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 $E \bigcup \{x\} B$, i.e. that $F \in B$. Now $D = B \bigcap PA$ 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 $C \bigcup \{x\} \in B$. 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

$$\bigcup [C \bigcup \{D\}] = [\bigcup C] \bigcup D$$

and $\bigcup C$ and D are both finite so $\bigcup [C \bigcup \{D\}]$ is finite. In other words, if $C \in B$ and $D \in A$ then $\bigcup [C \bigcup \{D\}] \in B$. 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 $E \bigcup \{x\} \in C$. 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 $G \bigcup \{x\}$ 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 $P[A \bigcup \{x\}] \setminus PA$ for any x. But

$$P[A \bigcup \{x\}] = PA \bigcup [P[A \bigcup \{x\}] \setminus PA]$$

and the union of two finite sets is finite so $P[A \bigcup \{x\}]$ 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 $B \bigcap C$ 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 $B \bigcup C$ is finite if B or C is finite.
- If A is finite then so is PA.

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.

Chains and pairs

A chain is a set A such that for all $B \in A$ and $C \in A$ we have $B \subseteq C$ or $C \subseteq B$. Any subset of a chain is also a chain. Every non-empty finite chain has a least member, i.e. a member which is a subset of every other member, and a greatest member, i.e. a member such that every other member is a subset of it. This can be proved by induction on sets. The least member is $\bigcap A$ and the greatest member is $\bigcup A$.

As an example, for any x and y the set $\{\{x\}, \{x, y\}\}$ is a chain, with $\{x\}$ as its least member and $\{x, y\}$ its greatest member. This is true even in the case $x = y$, although then $\{x\}$ and $\{x, y\}$ are the same set. As another example, for each natural number m we can consider the set of all natural numbers n such that $m \leq n$. The set of such sets, one for each m, form a chain. This chain has a greatest member, the set of all natural numbers, but no least element. That doesn't contradict the result above because this chain is not finite. This example, of course, assumes that the set of natural numbers exists.

Every subset of a chain is a chain and every subset of a finite set is a finite set so every subset of a finite chain is a finite chain.

Suppose A is a finite chain of non-empty sets and $D \in A$. Then

$$E = \{B \in A : [[B \subseteq D] \wedge [\neg[B = D]]]\},$$

i.e. the set of elements of A which are proper subsets of D, is a subset of A and so is a finite chain. It follows from what we showed earlier that if E is non-empty then it has a greatest member, which we'll call C, and $C = \bigcup E$. Now $C \in E$ so C is a proper subset of D. Therefore $D \setminus C$, which is the same as

$$D \setminus \bigcup \{B \in A : [[B \subseteq D] \wedge [\neg[B = D]]]\},$$

is non-empty. If E is empty then the set above is just D, which is a member of A and hence is also non-empty. So in either case the set above is non-empty. So for each $D \in A$ the set above has at least one member. We say that A is a Kuratowski chain if each of these sets also has at most one member. In other words, a set A is a Kuratowski chain if and only if it satisfies the following conditions.

- A is finite, i.e.,

$$[\forall B \in [PPA \setminus \emptyset] : [[\exists C \in B : [\forall D \in B : [[C \subseteq D] \supset [C = D]]]] \\ \wedge [\exists D \in B : [\forall C \in B : [[C \subseteq D] \supset [C = D]]]]]]$$

- A is a chain, i.e.

$$[\forall B \in A : [\forall C \in A : [[B \subseteq C] \vee [C \subseteq B]]]]$$

- Every member of A is a non-empty set, i.e.

$$[\forall B \in A : [\exists x.[x \in B]]]$$

- Each of the sets discussed previously has at most one element, i.e.

$$[\forall C.[C = D \setminus \bigcup\{B \in A : [[B \subseteq D] \wedge [\neg[B = D]]]\}]$$
$$\supset [[[x \in C] \wedge [y \in C]] \supset [x = y]]].$$

The set $\{\{x\}, \{x, y\}\}$ considered earlier is an example of a Kuratowski chain. It is called the Kuratowski pair with initial element x and final element y.

More generally, if A is a non-empty Kuratowski chain then we call the unique member of $\bigcap A$ the initial element of A and the unique element of

$$[\bigcup A] \setminus \bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\}$$

the final element of A. The fact that these sets have exactly one element is built into the definition of Kuratowski chains. The initial and final element could be the same, as happens, for example, with the chain $\{x\}$. This is the only way that can happen though.

Suppose A is a non-empty Kuratowski chain with at most two elements. Let x be the initial element and let y be the final element. Let C be the least member of A. By the definition of a chain the set

$$C \setminus \bigcup\{B \in A : [[B \subseteq C] \wedge [\neg[B = C]]]\}$$

has only one member. C is a least member of A so the set

$$\{B \in A : [[B \subseteq C] \wedge [\neg[B = C]]]\}$$

is empty and therefore

$$C \setminus \bigcup\{B \in A : [[B \subseteq C] \wedge [\neg[B = C]]]\} = C$$

and C has only one member. Since x is a member of all members of A and C is a member of A it follows that $C = \{x\}$. So

$$\{x\} \in A.$$

y is the final element of A so

$$[\bigcup A] \setminus \bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\} = \{y\}.$$

Now

$$\begin{aligned}\bigcup A = [[\bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\}] \\ \bigcup[[\bigcup A] \setminus \bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\}]\end{aligned}$$

so

$$\bigcup A = [\bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\}] \bigcup\{y\}.$$

A has at most two members so $[\bigcup\{B \in A : [[B \subseteq A] \wedge [\neg[B = A]]]\}]$ is either empty or has C as its only member. In the first case $x = y$ so $\bigcup A = \{y\}$ is the same as

$$\bigcup A = \{x, y\}.$$

In the second case $\bigcup A = C \bigcup\{y\}$ is the same as

$$\bigcup A = \{x, y\}.$$

So we get $\bigcup A = \{x, y\}$ in either case. For any Kuratowski chain $\bigcup A \in A$ so

$$\{x, y\} \in A.$$

We already had $\{x\} \in A$ so

$$\{\{x\}, \{x, y\}\} \subseteq A.$$

A has at most two members so

$$\{\{x\}, \{x, y\}\} = A.$$

Therefore all non-empty Kuratowski chains with at most two members are Kuratowski pairs. We've already seen the converse, that all Kuratowski pairs are non-empty Kuratowski chains with at most two members.

Ordered pairs

If two Kuratowski chains are equal then they must have the same initial element and same final element. The applies in particular to pairs. What

we've just shown implies that if two pairs have the same initial and final elements then they are equal.

Note that it's not meaningful to talk about the initial or final member of the two element set $\{x, y\}$ since $\{x, y\} = \{y, x\}$ but $\{\{x\}, \{x, y\}\}$ is not equal to $\{\{y\}, \{y, x\}\}$ unless $x = y$. Because we can uniquely identify an initial and final element Kuratowski pairs are often called ordered pairs.

It's tempting to try to define ordered triples analogously, so that $\{\{x\}, \{x, y\}, \{x, y, z\}\}$ would be the ordered triple with x as its initial element, y as its middle element and z as its final element. Unfortunately this doesn't work. Consider the ordered triple with v as both its initial and middle elements and w as its final element. According to the proposed definition above this would be $\{\{v\}, \{v, v\}, \{v, v, w\}\}$, which is the same as $\{\{v\}, \{v, w\}\}$. Now consider the ordered triple whose initial element is v and whose middle and final elements are w. According to the proposed definition above this would be $\{\{v\}, \{v, w\}, \{v, w, w\}\}$, which is also the same as $\{\{v\}, \{v, w\}\}$. These triples have distinct middle elements though and should therefore be distinct. So the Kuratowski construction works for pairs but the analogous construction for triples does not work.

Soon we will have chains whose elements are Kuratowski pairs, each of which has initial and final elements. To avoid confusion, or at least reduce it, from now on I'll refer to the initial element of a pair as the left component and the final element as the right component.

Lists

We can still build ordered triples, and lists more generally, using Kuratowski chains. We just have to go about it in a more complicated way.

We'll say that A is a list if it satisfies the following conditions:

- A is a finite Kuratowski chain
- for each $B \in A$ the unique member of the set

$$B \setminus \bigcup \{C \in B : [[C \subseteq B] \wedge [\neg [B = C]]]\}.$$

  is an ordered pair, the right component of which is

$$\{C \in B : [[C \subseteq B] \wedge [\neg [B = C]]]\}.$$

The items in the list are the left components of the pairs

$$B \setminus \bigcup \{C \in B : [[C \subseteq B] \wedge [\neg [B = C]]]\}.$$

These form a set.

The empty set $\emptyset$ is a list, called the empty list, with no items. The first item of a non-empty list is the left component of the initial element of the list. The last item of a non-empty list is the left component of the final element of the list. The definition of lists implies that if A is a list and B is its final element then the right component of B is a list with one element fewer than A. We'll call this the truncation of A. For any list A and any x there is a unique list whose last item is x and whose truncation is A, namely

$$B = A \bigcup \{\{\{x\}, \{x, A\}\}\}.$$

We say that B is the result of appending the item x to the list A. We can build up lists by starting with the empty set and successively appending items. The result of appending the items x, y, and z, in that order, to the empty set will be denoted (x, y, z), and similarly for any other list. The empty list is written as $()$.

The precise details aren't terribly important but I'll write out what the list (x, y, z) is as a set, using this technique of successively appending items.

$$() = \emptyset,$$

$$(x) = \{\{\{x\}, \{x, \emptyset\}\}\},$$

$$(x, y) = \{\{\{x\}, \{x, \emptyset\}\}, \{\{y\}, \{y, \{\{\{x\}, \{x, \emptyset\}\}\}\}\}\}$$

$$(x, y, z) = \{\{\{x\}, \{x, \emptyset\}\}, \{\{y\}, \{y, \{\{\{x\}, \{x, \emptyset\}\}\}\}\}\},$$
$$\{\{z\}, \{z, \{\{\{x\}, \{x, \emptyset\}\}, \{\{y\}, \{y, \{\{\{x\}, \{x, \emptyset\}\}\}\}\}\}\}\}$$

This looks rather complicated but all we really need to know is the following.

- There is a Boolean expression in our language which tells us whether something is a list.
- There is a Boolean expression in our language which tells us whether a list is empty.

- There is a way within our language to get the last item in a non-empty list.
- There is a way within our language to get a list with all but the last item.
- There is a way to take a list and append one more item at the end.
- There is a way to get the set whose members are the items of a list.

These are all the operations we need to perform on lists. More complicated operations, like reversing the order of the items on a list or concatenating two lists, can be defined from these basic operations. In fact LISP, whose main data structure is lists, does exactly this, with the inconsequential difference that it builds lists starting from the empty list by adding new items at the start of the list rather than the end.

Interfaces

A fundamental organising principle in programming is to define interfaces, the fundamental operations whose semantics the user of an object can rely upon, and to hide as much as possible the implementation of these interfaces, so that one implementation can be replaced by another without breaking anything, assuming users aren't relying on anything about the objects beyond the fact that they implement one or more of these interfaces.

As an example, we now have two different ways we could implement ordered pairs, either as Kuratowski pairs or as lists with two elements. The first option might seem more efficient, since

$$\{\{x\}, \{x, y\}\}$$

is simpler than

$$\{\{\{x\}, \{x, \emptyset\}\}, \{\{y\}, \{y, \{\{\{x\}, \{x, \emptyset\}\}\}\}\}\}.$$

This is irrelevant though if we only operate on lists using the primitive operations considered earlier, as we should. This is one respect in which mathematics differs from programming. Efficiency of implementation matters to users in programming. There is a notion of efficiency in mathematics but it involves the ease with which we can prove that the objects constructed

implement the given interface. Implementations are generally called definitions in mathematics.

In fact there are a number of other implementations of ordered pairs besides the two given above but we won't consider any of those.

Often one interface can be implemented in terms of another. If you look at the list of operations on lists defined above you may notice that by combining the third and fourth operations and removing the last one standard interface for a stack:

- We can check whether an object is a stack.
- We can check whether a stack is empty.
- We can pop the top item off a non-empty stack, leaving a stack with the remaining items.
- We can push an item onto the top of the stack, leaving a stack with that item as its top.

Which implementation of ordered pairs should we choose? Perhaps the most useful answer is that it shouldn't matter. If we ever do anything which works for one implementation but not the other then we are doing the mathematical equivalent of accessing objects outside of their public interfaces, which in programming is either forbidden or strongly discouraged, depending on the language. Still, we do need some implementation so we have to choose one. Efficiency, in the mathematical sense rather than the programming one, is not really an issue for us. We need lists for a number of other purposes, including the definition of a language, so we can't avoid defining lists and establishing their main properties, even if we ultimately decide not to define ordered pairs in terms of them. We need Kuratowski pairs for the implementation of lists above, so we also needs those definitions and properties. So the same amount of work is required no matter which option we choose. I'll use the list definition primarily because we need a notation for ordered pairs and a notation for lists and if ordered pairs are lists then we can use the same notation for both, so (x, y) is the two-element list whose first and last items are x and y respectively and also the ordered pair whose first and second coordinates are x and y, since those are the same thing. Mathematicians traditionally choose the other interpretation, as Kuratowski pairs, because they need ordered pairs

but don't really need lists. In that situation Kuratowski pairs are a more efficient implementation. As mentioned before though, it doesn't really matter.

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 can be proved fairly easily by induction on sets. 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 Δ_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 Δ_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 $R \circ \Delta_A = R$. If the range of R is a subset of A then $\Delta_A \circ R = R$.

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 $R \bigcap R^{-1}$ 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. $S \bigcap S^{-1} = R$ 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. $T \bigcap T^{-1} = \emptyset$ 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. $U \bigcap U^{-1} = U$ 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. $V \bigcap V^{-1} = V$ 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.

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 function from A to B then $F \circ F^{-1} = \Delta_B$ and $F^{-1} \circ F = \Delta_A$.

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 $A \bigcup \{x\}$ 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 $A \bigcup \{x\}$ 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 $z \in A \bigcup \{x\}$ 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 $A \bigcup \{x\}$

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 $z \in A \bigcup \{x\}$ 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

$$G = F \bigcup \{(w, y)\} \setminus \{(w, x), (x, y)\}.$$

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 $A \bigcup \{x\}$ is in the range of F, which therefore must be surjective. F was an arbitrary injective function from $A \bigcup \{x\}$ to itself so all injective functions from $A \bigcup \{x\}$ 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 $A \bigcup \{x\}$ to itself are surjective. By induction on sets it follows that all injective functions from a finite set to itself are surjective.

Order relations, equivalence relations

A relation R on a set A is called reflexive if $\Delta_A \subseteq R$, 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 $R \bigcup R^{-1} = A \times A$, 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.

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 $C = B \bigcup \{x\}$. 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 $C \bigcup \{x\} \in B$ for all $x \in A$. This is proved as follows. If $C = \emptyset$ then x is both a minimal and maximal member of $C \bigcup \{x\}$. 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 $y \in C \bigcup \{x\}$ and $(y, z) \in R$ imply $y = z$ so z is a minimal member of $C \bigcup \{x\}$. 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 $y \in C \bigcup \{x\}$ and $(y, x) \in R$ imply $y = x$. In other words x is a minimal member of $C \bigcup \{x\}$. So either x or z is a minimal member of $C \bigcup \{x\}$. A similar argument shows that $C \bigcup \{x\}$ 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 $S = R \bigcap R^{-1}$ 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.

Infinite sets

With the axioms above we have no way to prove the existence of an infinite set. The various operations we have, union, intersection, relative complement, power set and Cartesian product, all have the property that when applied to finite sets they produce finite sets.

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 Selection 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.

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.

Natural numbers

There are multiple ways to implement natural numbers within set theory.

The most common way is via the von Neumann ordinals. Consider the

operation $'$ on sets defined by

$$A' = A \bigcup \{A\}.$$

Then

$$\emptyset' = \{\emptyset\}, \quad \emptyset'' = \{\emptyset, \{\emptyset\}\}, \quad \emptyset''' = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, ...$$

In general the set represented by $\emptyset$ followed by n apostrophes has n members, each of which is one of the sets represented by $\emptyset$ followed by m apostrophes where m is a natural number less than n. We will call the sets of this form finite ordinals.

It is possible to define operations $+$ and $\cdot$ on sets in such a way that whenever B and C are finite ordinals we have the properties.

- $\{\forall B.[\neg(B' = \emptyset)]\}$
- $\{\forall B.[(B + \emptyset) = B]\}$
- $(\forall B.\{\forall C.[(B + C') = (B + C)']\})$
- $\{\forall B.[(B \cdot \emptyset) = \emptyset]\}$
- $[\forall B.(\forall C.\{(B \cdot C') = [(B \cdot C) + B]\})]$

The first of these doesn't reference the operations $+$ and $\cdot$ at all and is easily proved. B is a member of B' and $\emptyset$ has no members so B and $\emptyset$ cannot be equal. To prove the other four properties one needs the definitions of $+$ and $\cdot$, which are rather complicated. We won't do this. Instead we note that the properties listed above correspond exactly to the five axioms of Peano arithmetic. Peano arithmetic had rules of inference as well as axioms but the rules of inference also carry over. The most complicated of those rules of inference, the rule of induction, follows from the induction property of finite sets discussed earlier. In this way it is possible to build a model of Peano arithmetic entirely within simple set theory.

An alternative method to construct natural numbers is via lists. Choose some x. The choice doesn't matter but a convenient one is $x = \emptyset$, since we have an axiom which says it exists. Then a natural number is just a list all of whose items are x. The intuition is that the length of the list is a natural number and there is one such list for each natural number so we can use the lists as representatives for the number. From this point of view it's clear how we should define 0, incrementation and addition. 0 is just the

empty list `()`. A' is the result of appending an x to A. $A + B$ is the result of concatenating A and B. Multiplication is somewhat trickier to define but this can be done in such a way that the axioms of arithmetic are all satisfied.

The standard point of view in mathematics and logic is not that the finite ordinals have the same behaviour as the natural numbers but rather that they are the natural numbers. At least this is the point of view most mathematicians and logicians claim to have. It's debatable how many really believe this. One property of the definitions above is that $\emptyset' \in \emptyset'''$ or, in the usual notation for natural numbers $1 \in 3$. With the interpretation described above this is simply a theorem about the natural numbers but it would be hard to find a mathematician who would be comfortable describing $1 \in 3$ as a true statement, or even a meaningful one.

The situation here is similar to the one we encountered earlier with ordered pairs. A computer science perspective is more useful here than a mathematical or logical one. The natural numbers have an interface. Some operations, like $+$ and $\cdot$, are defined on them, as are some relations, like $=$ and $\leq$. These operations and relations are guaranteed to have certain properties, described by the Peano axioms.

Operators outside the public interface of the natural numbers, like $\bigcap$ and $\bigcup$ may behave differently in different implementations, as may relations like $\in$ and $\subseteq$. In practice once the implementation has been set up and the required properties have been proved one only ever uses the public interface. One doesn't, and shouldn't, write down statements like $1 \in 3$.

The fact that we can implement Peano arithmetic within set theory has an important consequence. Peano arithmetic is either inconsistent or incomplete. As a result set theory must also be either inconsistent or incomplete.

The set of natural numbers

It's important to understand what we have and haven't done above. We've found objects which behave like the natural numbers. We haven't shown that they form a set. In fact we can't show this with the axioms we have because these sets, if they exist, are infinite and our axioms are not sufficient to show the existence of any infinite set. We need a new axiom in order to have not just natural numbers but a set of natural numbers, which is what we need if we're to extend arithmetic beyond Peano arithmetic.

The simplest approach is just to assume these sets exist as an axiom.

For the von Neumann implementation of the natural numbers this means taking the following as an axiom:

- Infinity: There is a set A such that $\emptyset \in A$ and and $B \bigcup \{B\} \in A$ whenever $B \in A$. Formally,

$$[\exists A.[[\emptyset \in A] \wedge [\forall B \in A : B \bigcup \{B\} \in A]]].$$

The set B is not necessarily the set N that we're looking for. As with a number of previous axioms we've assumed the existence of a set which is large enough to contain everything we want, but might contain other things. To remove those we use the axiom of separation, selecting only those $B \in A$ which are finite and satisfy the condition

$$[\forall C \in B : [\forall D \in C : [[D \in B] \wedge [\forall E \in D : E \in C]]]].$$

If we use the list implementation of the natural numbers then it's more natural to use the following axiom

- Infinity (alternative version): For every set C there is a set D whose members are the lists all of whose items are members of C.

The formal version of this axiom is rather long since it needs to incorporate the definition of a list.

Although these axioms were designed for the same purpose they are not quite equivalent, in the sense that we can't prove either from the other and our other axioms and rules of inference. They become equivalent though if you assume the following additional axiom schema.

- Replacement: For any Boolean expression P in which x, y, and A appears freely but B does not the statement that for each A if $x \in A$ implies that there is a unique y such that P then there is a set B such that $y \in B$ if and only if there is an $x \in A$ such that P. Formally this is

  $[\forall A.[[\forall x \in A : [[\exists y.P] \wedge [[P \wedge Q] \supset y = z]]]$
  $\supset [\exists B.[\forall y.[[[y \in B] \supset [\exists x \in A : P]] \wedge [[\exists x \in A : P] \supset [y \in B]]]]]]].$

  Here z is a variable which does not occur in P and Q is the result of replacing all free occurrences of y in P with z.

The Axiom of Replacement can be understood as follows. If we have sets A and B and a Boolean expression involving variables x and y then we can form a relation from A to B which is the set of pairs (x, y) where $x \in A$ and $y \in B$. There are some conditions, i.e. Boolean expressions, which need to be satisfied for this relation to be a surjective function, in which case A is its domain and B is its range. The Axiom of Replacement says that if A is a set and we have such a Boolean expression satisfying those conditions then there is indeed a set B such that the relation defined as above is a surjective function from A to B.

The Axiom of Replacement was not part of Zermelo's set theory. It was introduced later by Fraenkel. It is often convenient, but rarely necessary, to assume it in order to prove standard mathematical theorems. It forms part of what's called Zermelo-Fraenkel set theory, which is the version of set theory most mathematicians use.

Since it's rarely used I don't want to spend too much time on it but I will sketch how you can prove the alternative version of the Axiom of Infinity from the first version and the Axiom of Replacement.

One nice property of both implementations of the natural numbers is that the natural number n is a set with n members. With either implementation we can therefore prove that for every finite set there is a bijective function from that set to a natural number, considered as a set. This is in fact fairly straightforward to prove by set induction.

Since we're assuming the first version of the Axiom of Infinity we'll use the implementation of the natural numbers as von Neumann ordinals, so that the set of natural numbers is known to exist. We can write down a Boolean expression with free variables x and w expressing the following:

- x is a natural number, i.e. a von Neumann ordinal, and
- w is a list all of whose items are members of C,
- there is a bijective function from x to w.

Each of these statements individually is straightforward, if rather tedious, to express in our language and we just need to combine them with $\wedge$'s. Using this combined Boolean expression we can construct a second expression, with free variables x and y expressing the fact that y is a set and $w \in y$ if and only if the first expression is satisfied. This second expression then

has the interpretation that y is the set of all lists with x items, all of which are members of C. We then apply the Axiom of Replacement with $A = N$ and P being the Boolean expression we just constructed. This gives us a set B whose members are the y's make the expression true for some $x \in N$. In other words they are the sets of lists of each length. Every list is a finite set so by the theorem from the preceding paragraph it's a member of one of these sets. So the set of all lists of items in C is $D = \bigcup B$.

There's a similar, slightly easier argument which shows that the alternative form of Axiom of Infinity, together with the Axiom of Replacement, implies the first form.

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 Δ_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 $A \bigcup \{x\} = A$ and so F is also an injective function from $A \bigcup \{x\}$ 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 $A \bigcup \{x\}$ to N. So in either case there is an injective function from $A \bigcup \{x\}$ to N and so $A \bigcup \{x\}$ is countable. So $\emptyset$ is countable and if A is countable then so is $A \bigcup \{x\}$. 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 $B \bigcap C$ 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 $A \bigcup B$ to N so $A \bigcup B$ is countable.

N itself is countable since Δ_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.

Axiom(s) of choice

There are a variety of axioms with similar names, all of which involve the word choice in some way. It's unclear whether any of them are really needed for Computer Science. Large parts of mathematics also don't require any of them but certain subjects need at least the Axiom of Dependent Choice in order to prove some of their main theorems. There is a stronger axiom, known just as the Axiom of Choice which is often assumed. As we'll see though, it has some disturbing consequences.

Computational paths

One way to think of the Axiom of Dependent Choice is in terms of the computational paths of non-deterministic state machines. We'll consider a particular simple type of such machines. These do not read any input. They have a well defined initial state and for each possible state there are one or more possibilities for the next state. Because there is at least one possibility in each state the computation will never terminate. To model such a machine we need a set A of possible states, a particular $w \in A$ to serve as the initial state and a relation T on A describing the possible state transitions. More precisely $(x, y) \in T$ if and only if the machine can transition from x to y in a single step. The condition that there is at least one possible transition from each state is then equivalent to the statement that T is left total. The machine is deterministic if there is also at most one possible transition from each state, i.e. if T is right unique. In this case it is a function from A to A.

A simple deterministic state machine of this kind can compute the pow-

ers of two. We just take $A = N$, $w = 1$ and take F to be the set of pairs of the form $(x, 2 \cdot x)$. A more interesting example is the Fibonacci sequence. This might not seem to fit the pattern described above because the n'th Fibonacci number depends on the $n-1$'st and the $n-2$'nd. This is easily fixed though. we take $A = N^2$ and take $w = (0, 1)$. T is the set of pairs of the form $((j, k), (k, j + k))$. The left element of the state after n steps is then the n'th Fibonacci number.

By a computational path we will mean a sequence, finite or infinite, of states through which the process can proceed starting from the initial state and making only those transitions allowed by the transition relation. More formally, it's a function F whose domain is either all of N or a subset of the form $\{0, 1, \ldots, n\}$ with the properties that

- $(0, w) \in F$
- If $(m, x) \in F$ and $(m + 1, y) \in F$ then $(x, y) \in T$.

The interpretation is that $(m, x) \in F$ means that the machine is in the state x after m steps. The first of the statements above expresses the condition that the initial state is w while the second expresses the condition that the transitions from one state to the next are only those allowed by T. We also need to express the condition on the domain of F stated above. The simplest way to do this is as follows.

- If $m \leq n$ and there is a $y \in A$ such that $(n, y) \in F$ then there is an $x \in A$ such that $(m, x) \in F$.

This says that if n is in the domain of F then so are all smaller numbers. Finally we need to express the fact that F is right unique, i.e. that the machine can only be in one state at a given time.

- For any $n \in N$, $x \in A$ and $y \in A$, if $(n, x) \in F$ and $(n, y) \in F$ then $x = y$.

A computational path is any subset of $N \times A$ satisfying the conditions above. I'll denote the set of all computational paths by B.

The first thing I'll show is that there is no upper bound on the length of computational paths. More precisely, for any $n \in N$ there is an $F \in B$ and an x in A such that $(n, x) \in F$. This is proved by induction on n. For $n = 0$ it's true because $x = w$ and $F = \{(0, w)\}$ has the required properties. If it's

true for n then we can prove it for $n+1$ as follows. Let G be the set consisting of (k, z) where either $k \leq n$ and $(k, z) \in F$ or $k = n+1$ and $z = y$, where y is such that $(x, y) \in T$. We know there is such a y because T is left total. Then $G \in B$ and $(n+1, y) \in G$. This establishes the inductive step.

It would also have been possible to use set induction rather than arithmetic induction above but arithmetic induction is probably more familiar.

If F and G are both members of B their union $F \bigcup G$ may fail to be a member for a variety of reasons, including not being right unique. This can't happen for deterministic machines though. If T is right unique, $F \in B$ and $G \in B$ then $F \bigcup G \in B$. The hardest part of showing this is the proof that $F \bigcup G$ is right unique. This is proved by induction. More precisely, we show by induction on n that if $(n, v) \in F$ and $(n, x) \in G$ then $v = x$. For $n = 0$ this is certainly true because then we must have $v = w$ and $x = w$. If it's true for n then it's true for $n+1$. To see this we note that if $(n+1, z) \in F$ and $(n+1, y) \in G$ then $(v, z) \in T$ and $(x, y) \in T$. But $v = x$ so the assumed right uniqueness of T implies that $y = z$.

Still assuming that T is right unique, we have $\bigcup B \in B$. Again the right uniqueness is the hardest part to prove. If $(n, v) \in \bigcup B$ and $(n, x) \in \bigcup B$ then there must be $F \in B$ and $G \in B$ such that $(n, v) \in F$ and $(n, x) \in G$. But we've just seen that this implies $v = x$, so $\bigcup B$ is right unique.

We've already seen that for every $n \in N$ there is an $F \in B$ and and $x \in A$ such that $(n, x) \in F$. We then also have $(n, x) \in \bigcup B$. So n is a member of the domain of $\bigcup B$. This is true for all $n \in N$ so the domain of $\bigcup B$ is therefore all of N. Thus $\bigcup B$ is a computational path of infinite length. Previously we knew that there was no upper bound on the lengths of computational paths but we didn't know there was a computational path of infinite length.

Compared to a deterministic state machine a non-deterministic state machine offers at least as many options at each step so it might seem intuitive that if every deterministic state machine has an infinite computational path then so does every non-deterministic one. The proof above used the assumption that T is right unique in an essential way though.

In the case where A is countable one can still prove the existence of infinite computational paths without the assumption that T is right unique. If A is uncountable then this is no longer possible. If we want the statement to be true then we need to add it as an axiom.

Dependent choice

Our new axiom is as follows.

- Dependent Choice: For every set A, member $w \in A$ and left total relation T on A there is a function F from N to A such that if $(n, x) \in A$ and $(n + 1, y) \in A$ then $(x, y) \in T$.

As you can see, this is an exact translation of the statement that non-deterministic state machines of the type considered in the previous section have infinite computational paths.

This formulation of the axiom, which is the standard one, has an intuitive appeal but it's formalisation is quite complicated because it has buried in it notions like functions and relations and the set of natural numbers. There are equivalent axioms which are less intuitively appealing but easier to formalise. The following simple one appears to be due to Wolk:

- Dependent Choice (alternative formulation): If every chain in a partially ordered set is finite, then it contains a maximal element.

By chain here we a mean subset such that the partial order, when restricted to the subset, is a total order. You can check that this usage is consistent with the definition of Kuratowski chains given earlier, in the sense that every Kuratowski chain is a chain, with the partial order being given by set inclusion.

It's worth noting that for the application to state machines we only really need the Axiom of Dependent Choice when the set of states is uncountable. If A is countable and T is a left unique function from A to A we can construct a function F from A to A as follows. The countability of A means there is an injective function G from A to N. For any $x \in A$ the set of $z \in N$ for which there is a $y \in A$ with $(x, y) \in T$ and $(y, z) \in G$ is non-empty, since T and G are left total, and so has a least member. If we call this least member z then there is only one $y \in A$ such that \${ $(y, z) \in G$, since G is injective. We take (x, y) to be a member of F, and do not take any other ordered pair whose left component is x. Then F is a function and $F \subseteq T$. Because F is a function the deterministic state machine for which it is the transition relation must an infinite computional path, even without assumine the Axiom of Dependent Choice. Because F is a subset of T any computational path for this deterministic state machine is also a valid computational path for the

non-deterministic machine with transition relation T. There aren't many situations in computer science where you'd want to consider a state machine with uncountably many states. There are quite a few in mathematics though.

The Axiom of Choice

Zermelo had one further axiom, which is not generally included in what we now call Zermelo-Fraenkel set theory. This is the Axiom of Choice. In his initial formulation this axiom said that for any set of disjoint non-empty sets there is a set which has precisely one member from each of those sets. The restriction to non-empty sets is clearly necessary since you can't have precisely one member from a set with no members.

It turns out that there are many different axioms one could take which are equivalent to this one, in the sense that if one assumes any one of them as an axiom then the rest all become theorems. Some of these formulations have more intuitive appeal than others. Zermelo's version at least has the advantage that it's clear why it's named the Axiom of Choice. The set whose existence it asserts is obtained by choosing one element from each of the sets in our set. There are whole books devoted to equivalents of the Axiom of Choice.

The Axiom of Choice is known to give a consistent formal system, at least if we assume that Zermelo-Fraenkel is a consistent system. It's also known not to be a theorem in the Zermelo-Fraenkel system, assuming ZF is consistent. In other words one can prove that it can't be proved. The same, of course, applies to any of the statements known to be equivalent to the Axiom of Choice.

For the Axiom of Dependent Choice there was an equivalent statement in terms of chains. There is one for the Axiom of Choice as well. It goes by the name of Zorn's lemma, even though it is often taken as an axiom in place of the Axiom of Choice.

- Zorn's Lemma: If every chain in a partially ordered set is bounded, then it contains a maximal element.

A subset B of a set A with a partial order R is said to be bounded if there is a $y \in A$ such that $(x, y) \in R$ for all $x \in B$. This might look like the defini-

tion of a greatest element but there is a crucial difference. We only require y to be a member of A, not of B. An example of a bounded subset is the set of rational numbers x such that $0 < x$ and $x < 1$. this is a bounded subset of the rationals, with the partial order given by $\leq$, because $x \leq 1$ for all such x. In fact this set is also a chain. You may wonder whether this example contradicts Zorn's lemma, since the rationals do not have a maximal element. It doesn't. The hypothesis in Zorn's lemma is that every chain is bounded. This particular one is but some others are not. N, the set of natural numbers, is an example of an unbounded chain.

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 $B_1 \bigcup B_2$.
- 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

$$[\forall A.[[\exists B.B \in A] \supset [\exists C \in A : [A \bigcap C] = \emptyset]]].$$

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
- Separation
- Power Set

- Union
- Infinity, in the first of the two versions I presented
- Replacement
- Foundation
- possibly 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.

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 has no possible application 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 the second of the two versions I presented

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, i.e. calculus, number theory, etc.

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.

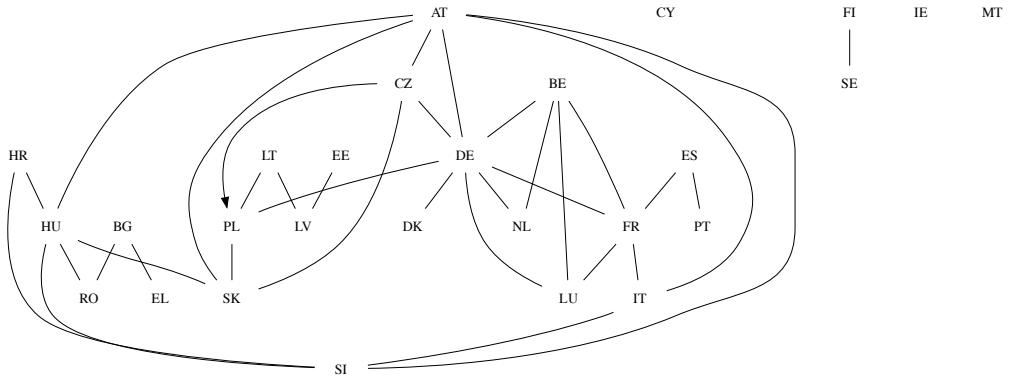


Figure 10: An undirected graph

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 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

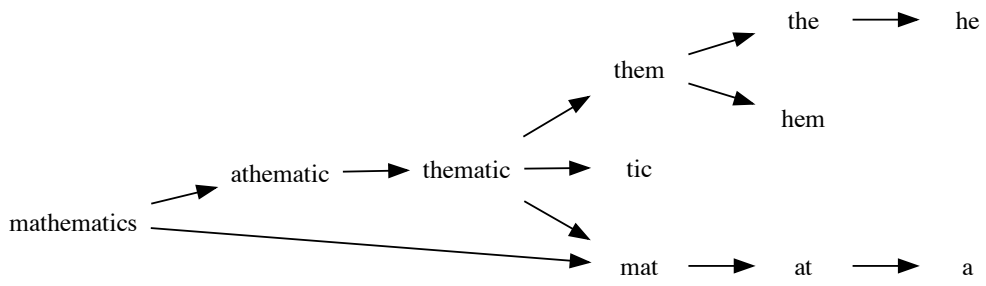


Figure 11: A directed graph

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.

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:

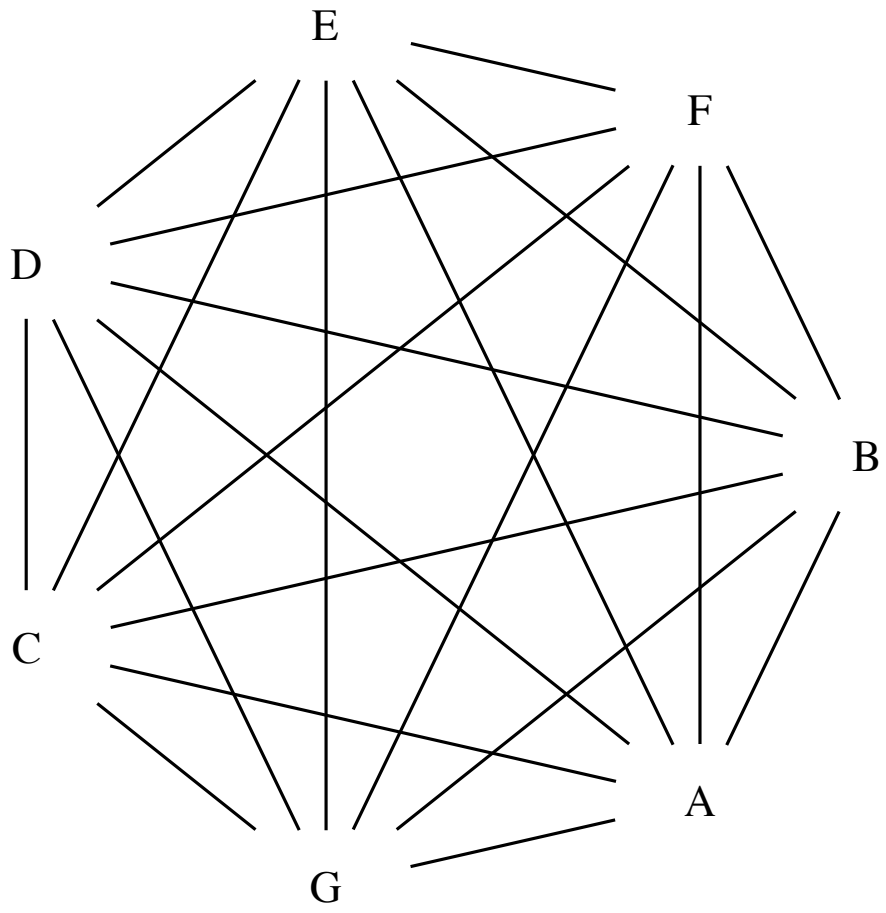


Figure 12: Another undirected graph

- 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.

We'll stick to finite graphs. This covers the graphs associated to finite state automata, abstract syntax trees, and computational paths of pro-

cesses guaranteed to terminate. It excludes computational paths of some non-terminating processes, which is unfortunate, but the theory of finite graphs is much simpler.

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 graphs with no self loops.

Definition

With the conventions chosen above we can define a graph as a finite 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 the study of relations on finite sets, but the questions we ask when considering such a relation as a graph are different from the ones we normally ask about relations.

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).

This definition is easily adapted to infinite graphs–you just drop the assumption that the set of vertices is finite–but it would require more serious modifications to cope with multiple edges from one vertex to another.

Ways to describe 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 graphs. As an example, consider the graph whose diagram has vertices labelled a to e and edges labelled 1 to 6.

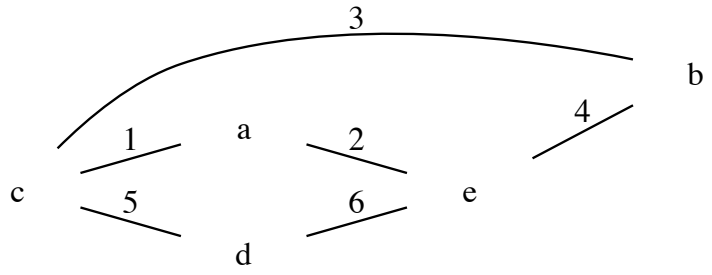


Figure 13: An undirected graph with labelled edges

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

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.

Bipartite graphs arise in a variety of contexts. Consider, for example, a graph whose vertices are the possible ways to list the members of a finite set with edges between lists which differ only by swapping the order of a pair of items. This is shown in the diagram. For simplicity I've chosen the set of the four letters a, b, c, and d, and have omitted the parentheses and commas from our usual list notation, so (a, b, c, d), for example, is represented by $abcd$.

It's not obvious staring at this graph that it's bipartite. One way to show this would be to use different coloured labels for the vertices in the two different sets. Another way to show it is to draw a directed graph which has one edge from each pair in the graph above, chosen consistently to that the set goes from a vertex in one set to a vertex in the other.

In this case, with four members of our set, the graph happens to be a $K_{6,6}$, but with more members the graph would not be $K_{p,q}$ for any choice of p and q.

Isomorphism

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

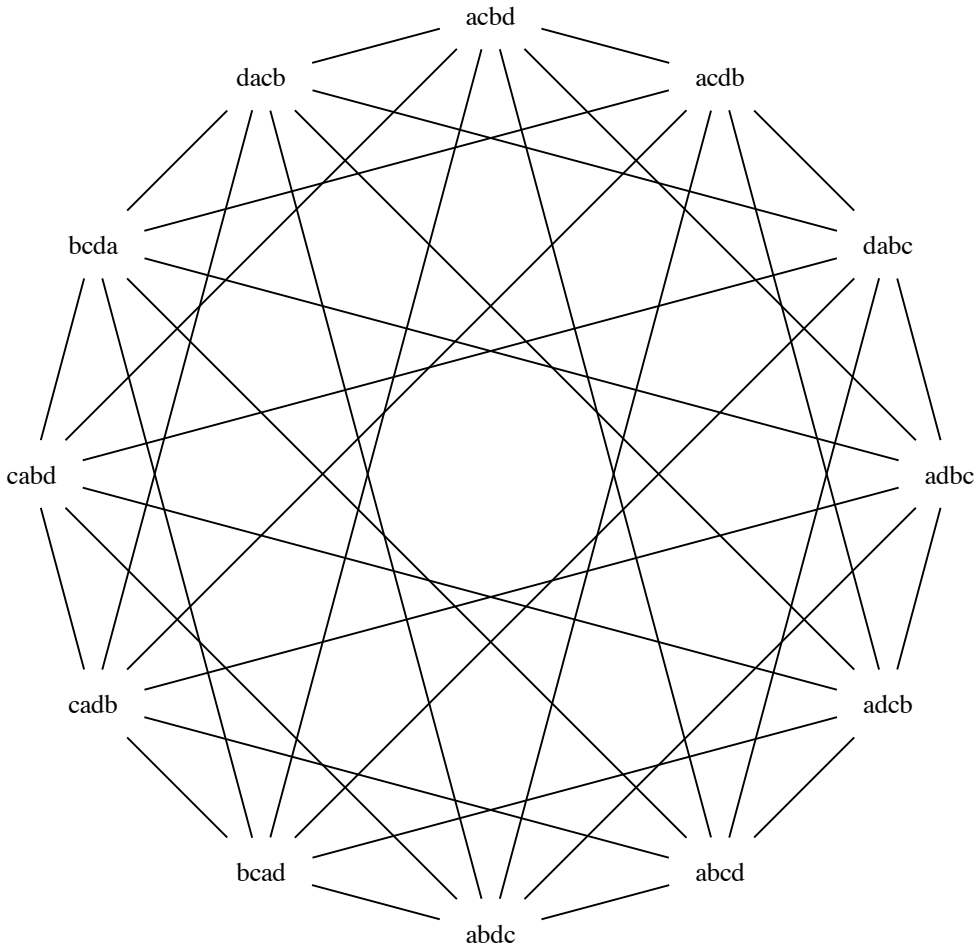


Figure 14: A bipartite graph

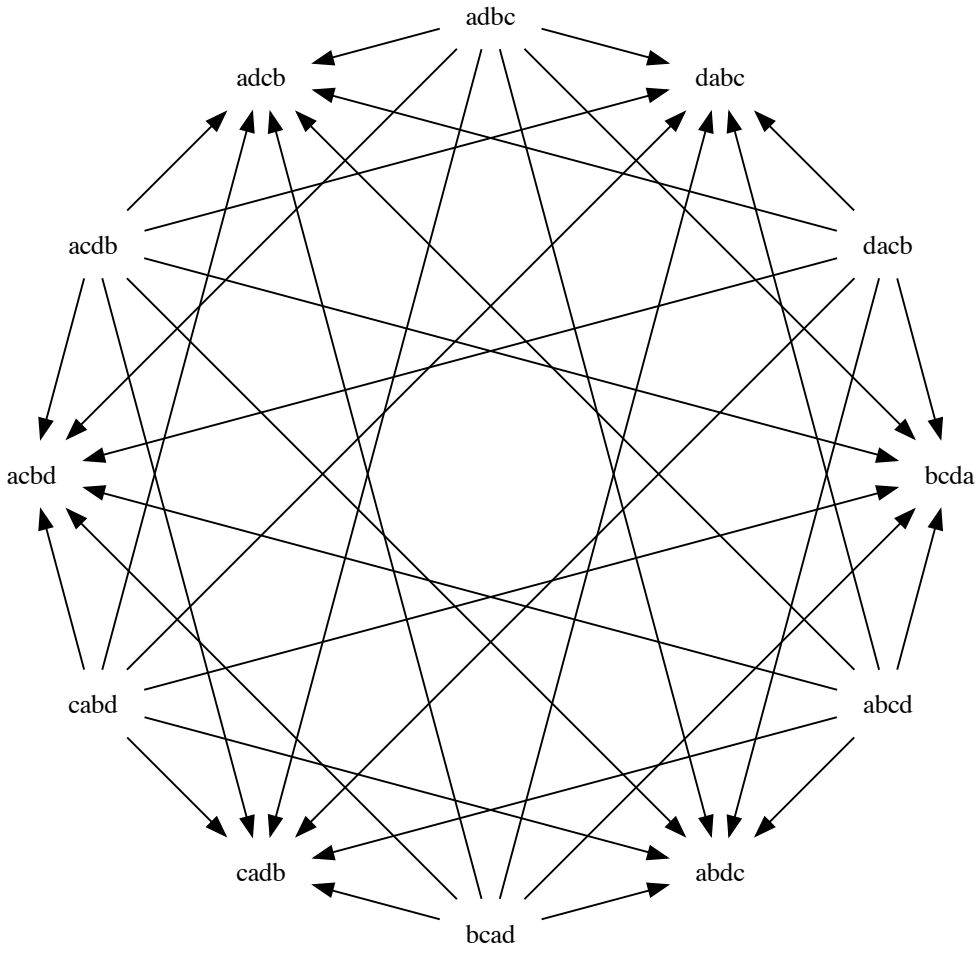


Figure 15: The corresponding directed graph

V to W is called a graph 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. The $K_{6,6}$ graph we just saw has a number of automorphisms which are geometrically obvious, such as rotations through any integer multiple of 30 degrees, but it has many more. We can permute the members of each group of six vertices separately. We can also rotate through 30 degrees, which switches the two groups, and then permute within each of them. There is a binary choice whether to rotate first and then $6! = 720$ possible permutations within each group, for a total of $2 \cdot 720 \cdot 720 = 1,036,800$ automorphisms. I won't list them.

More generally, $K_{p,q}$ 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.

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 whose vertices are the set of all lists of length four with items a, b, c, and d, i.e. the same vertices as in the example above, but with edges from each vertex to every other vertex. This is a K_{12}. The graph considered earlier, which we saw was a $K_{6,6}$, is a subgraph of it. More generally, any $K_{p,q}$ is a subgraph of K_{p+q}.

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.

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.

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

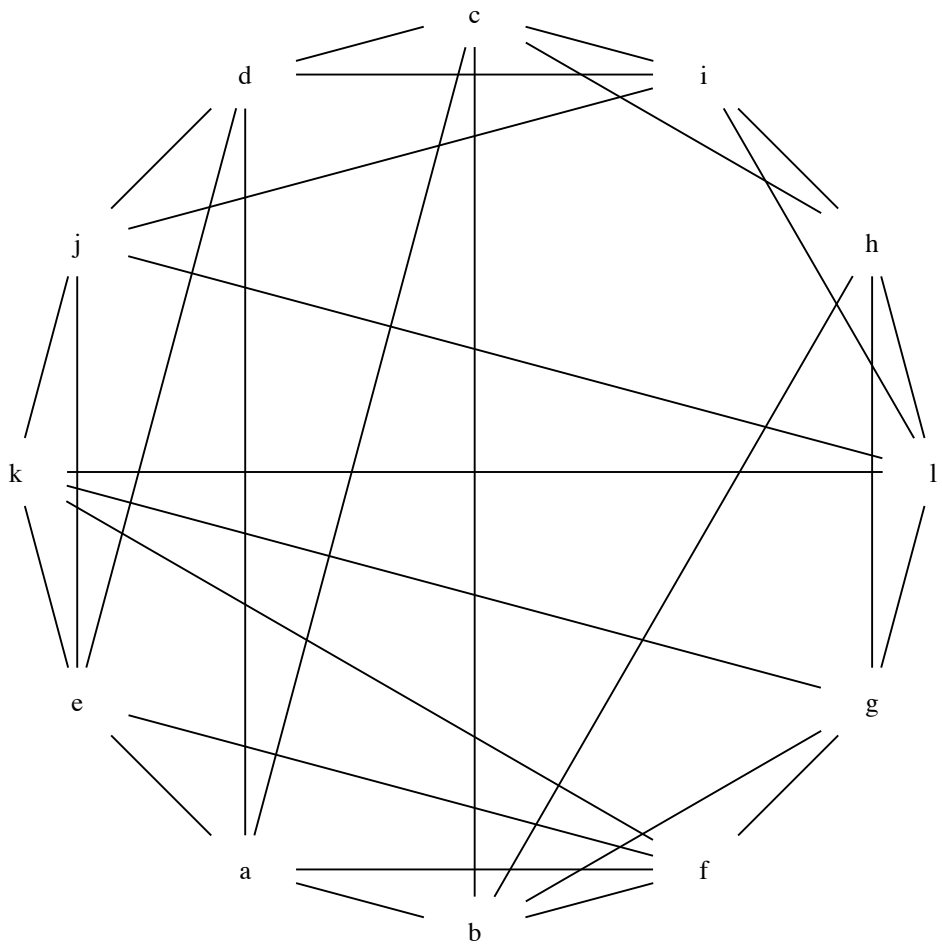


Figure 16: A regular graph

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.

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. Our bipartite graph example will work. This is a regular 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 *abcd*, *cadb*, *acdb*, *dabc*, *abcd*, *abdc*, *bcad*, *cadb*, *cabd*, *adcb*, *dacb*, *abdc*, *cabd*, *acbd*, *acdb*, *adcb*, *adbc*, *bcda*, *bcad*, *acbd*, *adbc*, *dabc*, *dacb*, *bcda*, *abcd*, *acbd*, *dacb*, *cadb*, *adbc*, *abdc*, *acdb*, *bcda*, *cabd*, *dabc*, *bcad*, *adcb*, and *abcd* in that order.

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. We found an Eulerian path in our bipartite graph earlier. The corresponding directed graph is given in the accompanying figure.

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

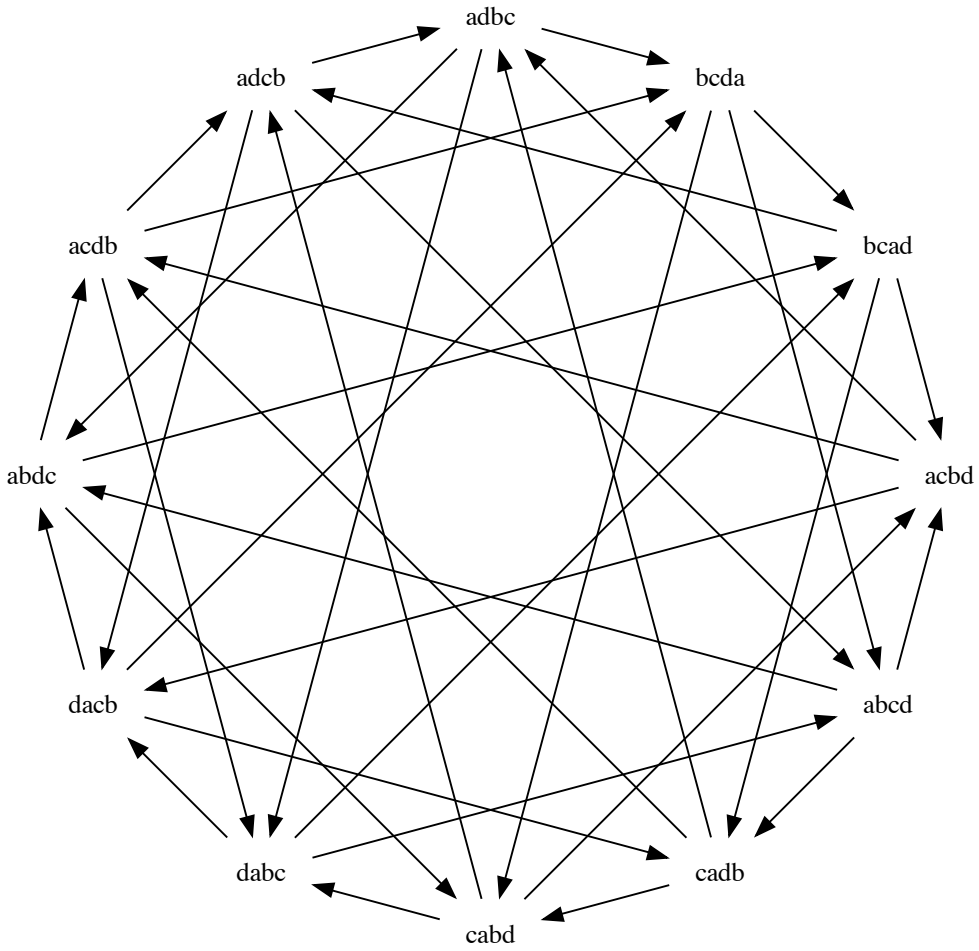


Figure 17: An Eulerian path in a bipartite graph

vertex arise in one of the ways just described. So for all but the initial and final 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 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.

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.

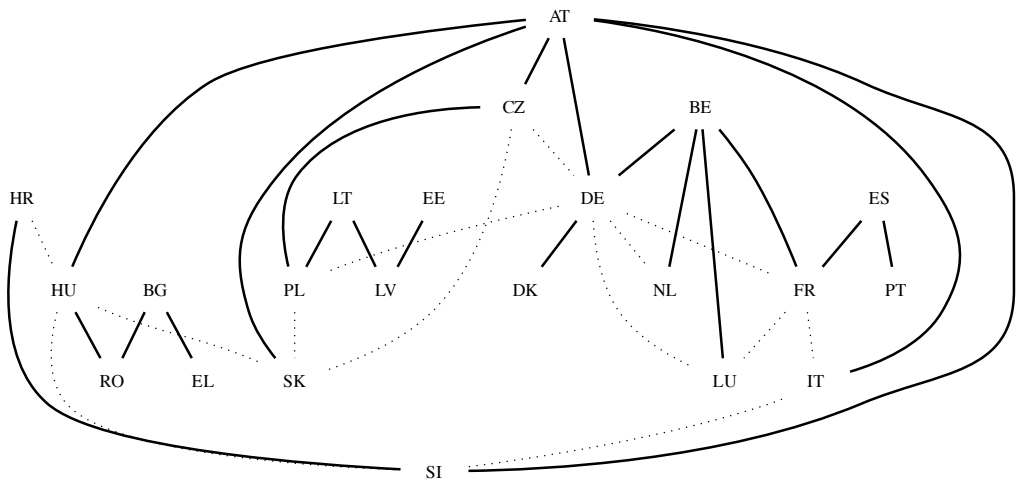


Figure 18: An undirected graph

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
- $\bigcap$ on the power set of a given set

- $\bigcup$ 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 itelf
- $\circ$ on the set of relations on a given set to itelf
- 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 $x \cdot y$, $A \bigcup B$, 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

$$A \bigcup (B \bigcup (C \bigcup D)) = (A \bigcup B) \bigcup (C \bigcup D) = ((A \bigcup B) \bigcup C) \bigcup D.$$

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.

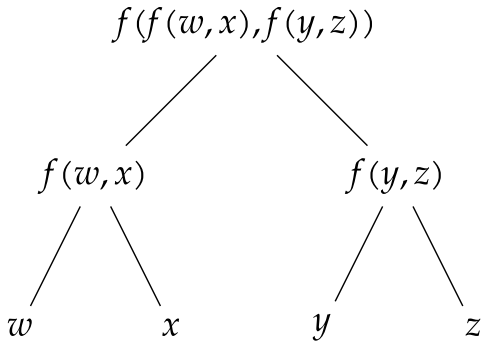


Figure 19: A tree for $f(f(w,x),f(y,z))$

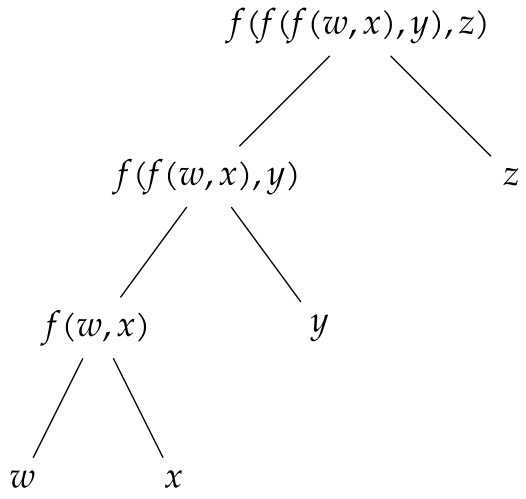


Figure 20: A tree for $f(f(f(w,x),y),z)$

We see that our earlier argument, based on applying the associative property twice, showed that three of these are equal. A similar argument shows

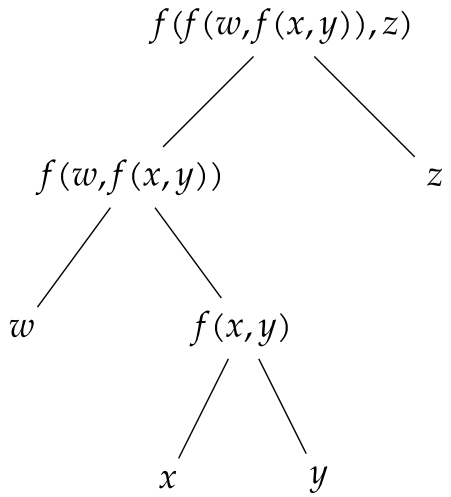


Figure 21: A tree for $f(f(w,f(x,y)),z)$

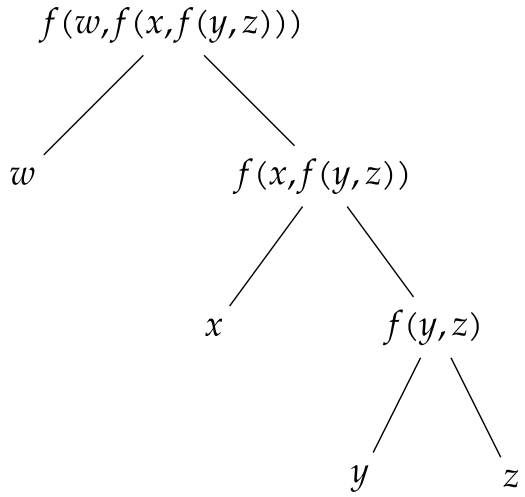


Figure 22: A tree for $f(w,f(x,f(y,z)))$

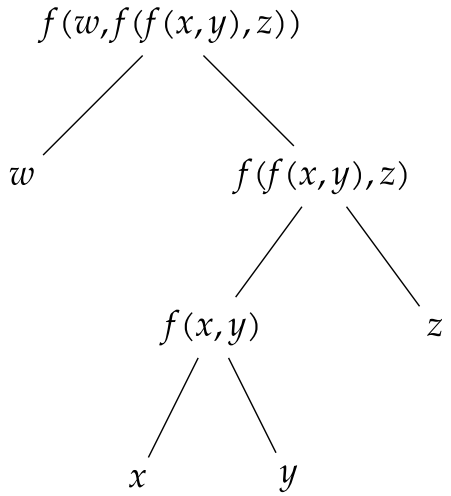


Figure 23: A tree for $f(w,f(f(x,y),z))$

that the other two are also equal to these. We'd like a general theorem, though, applying to all values of n.

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 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.
- $\bigcap$ on the power set of a given set: the whole set is an identity element.
- $\bigcup$ 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 setf: 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

```
%start weird

%%

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 capa-

ble 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

```
%start weird

%%

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:

```
%start integer

%%

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 regular 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

```
%start union

%%

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.

```
%start intxxyy
```

```
%%

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 it's 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

```
%start intxxyy

%%

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.

```
%start intxxyy

%%

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

```
%start intxxyy

%%

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.

```
%start weirdl

%%

weirdl : xxyyl
       ;
```

```
xxyyl  : | x xxyyl | y yyl
       ;

yyl    : | y yyl | x errorl
       ;

errorl : x errorl | y errorl
       ;
```

and

```
%start weirdr

%%

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.

```
%start weirdl

%%

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`.

```
%start weirdl

%%

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.

```
%start weirdl

%%

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 start

%%

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

```
%start weird

%%

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

```
%start weird

%%

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

```
%start ksxy

%%

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.

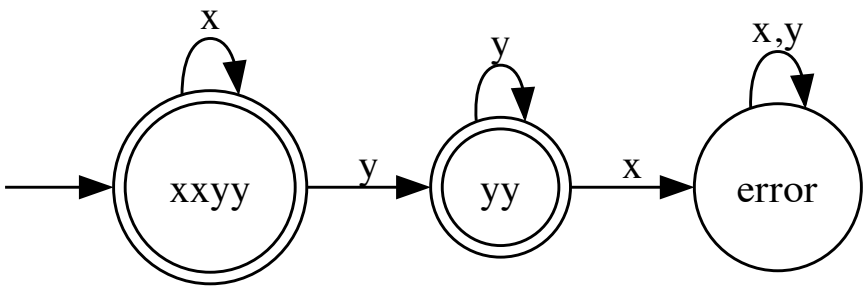


Figure 24: A finite state automaton for the the xy language

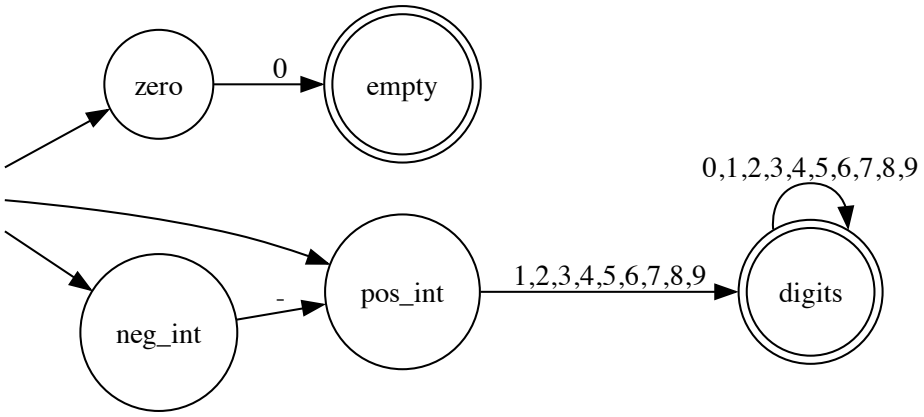


Figure 25: A finite state automaton for the the integer language

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 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 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.
- $F' = \{B \in PS : B \bigcap F \neq \emptyset\}$, 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. Those automata always reach the end of their input and are 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 $L_1 \bigcup L_2 \in R$.
- 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 $\bigcap S$.

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.

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 representable 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.

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 finite 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 pos-

sible 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 pump-

ing 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.

An example

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.

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.

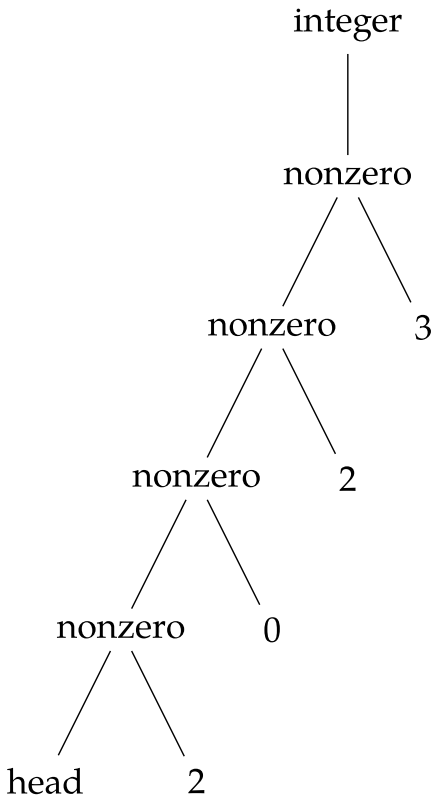


Figure 26: Parse tree for the string 2023

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.

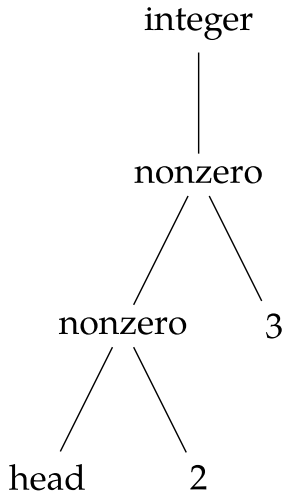


Figure 27: Parse tree for the string 23

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.

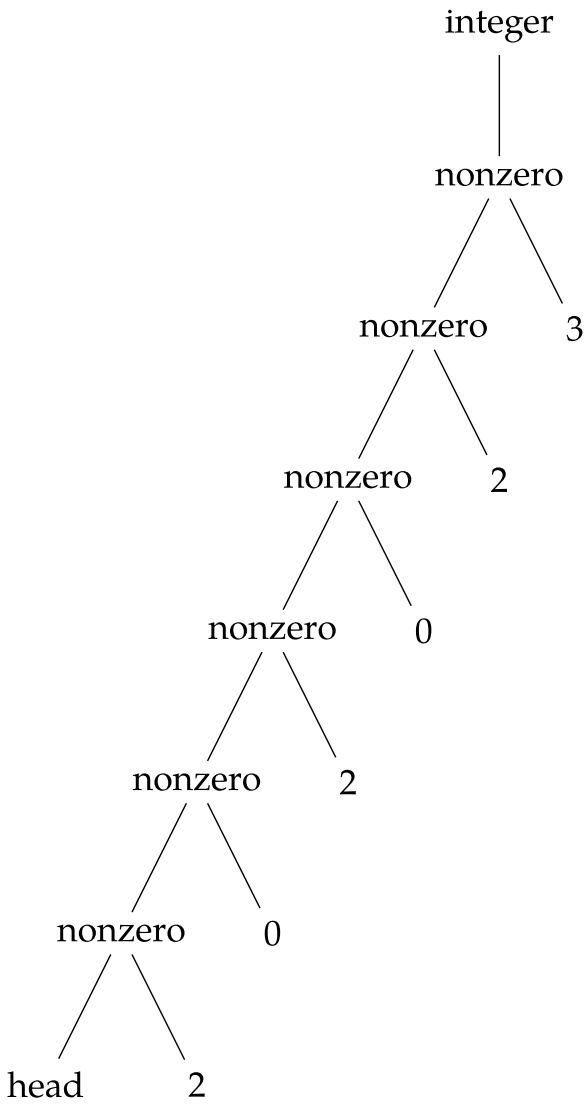


Figure 28: Parse tree for the string 202023

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

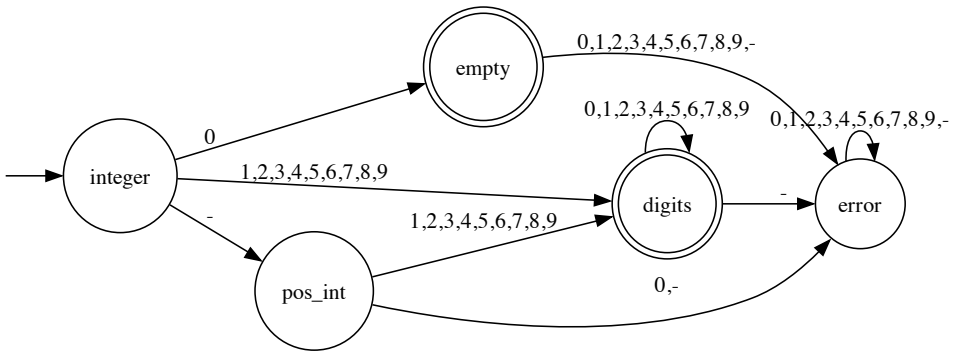


Figure 29: A strongly deterministic finite state automaton for the integers

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 autmaton 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. I won't give the construction here though.

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 occurences of it. We'll call the one closer to the root the outer occurence and the one farther from the root the inner occurence. Let c be the expansion of the inner occurence and let f be the expansion of the outer occurence. 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 occurence is at most $s + 1$. Taking the parse tree for w and replacing the part of the tree descending from the outer occurence with the part of the tree descending from the inner occurence 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 occurence with the part descending from the outer occurence, 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.